Università degli Studi di Padova
Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingeneria Informatica

tesi di laurea

LibEXI

Analisi e implementazione del formato EXI su reti di sensori wireless

Relatore: Michele Zorzi
Correlatore: Angelo P. Castellani

Laureando: Mattia Gheda

28 Giugno 2010

A mio Padre

Sommario

In questo documento è descritta l'implementazione dello standard EXI per architetture con forti limitazioni hardware.
Il funzionamento è stato testato su una rete di sensori realizzata con nodi TMoteSky della Berkeley University e TelosB della Crossbow Technology.
La libreria risultante è stata integrata all'interno del framework del progetto SENSEI.

Autore: Mattia Gheda

Indice

Sommario

1 Introduzione **1**
- 1.1 Scenario . . . 1
- 1.2 XML e EXI . . . 1
- 1.3 SENSEI . . . 2
- 1.4 Contributo dell'autore . . . 4

2 Piattaforma Hardware **7**
- 2.1 Wireless Sensor Network . . . 7
- 2.2 Telosb/TMote Sky . . . 9

3 Piattaforma Software **11**
- 3.1 Il linguaggio Ruby . . . 11
  - 3.1.1 Cenni storici . . . 12
  - 3.1.2 I Principi di base Ruby . . . 13
- 3.2 TinyOS, C e NesC . . . 13
  - 3.2.1 NesC . . . 14
  - 3.2.2 C . . . 15
  - 3.2.3 TinyOs . . . 15
- 3.3 Il codice del progetto SENSEI - Scenario Applicativo . . . 16

4 Lo standard EXI **21**
- 4.1 XML e grammatiche . . . 22
- 4.2 Lo Stream EXI . . . 24
- 4.3 EXI come flusso di eventi . . . 26
- 4.4 Eventi e grammatiche . . . 27
- 4.5 Il processo di codifica . . . 29
- 4.6 Codifica dei tipi . . . 30
  - 4.6.1 Esempio di algoritmo per la decodifica di interi senza segno: 32
- 4.7 EXI grammars . . . 32

4.8 Generazione delle Grammatiche . . . 34
4.8.1 Passo 1 . . . 36
4.8.2 Passo 2 . . . 36
4.8.3 Passo 3 . . . 37
4.9 Il processo di Codifica . . . 38

5 L'applicazione sviluppata 43
5.1 Assunzioni iniziali . . . 44
5.2 Scelte implementative . . . 45
5.2.1 Il formato della memoria . . . 47
5.2.2 Il formato delle grammatiche . . . 50
5.3 Il preprocessore Ruby . . . 52
5.3.1 Schema di lavoro . . . 53
5.3.2 Generazione Grammatiche . . . 54
5.3.3 Generazione Struct . . . 55
5.3.4 Generazione C grammars . . . 56
5.3.5 XML helper . . . 58
5.4 Il processore C . . . 58
5.4.1 Lo stack di automi . . . 60
5.4.2 Le funzioni di utilizzo della memoria (API) . . . 62
5.4.3 Le funzioni di codifica/decodifica dei tipi . . . 65
5.5 Ottimizzazioni . . . 66
5.5.1 Grammatiche . . . 66
5.5.2 Parametri . . . 67
5.5.3 Analisi del codice . . . 67
5.5.4 Attivatori di funzione . . . 67

6 Risultati e Conclusioni 69
6.1 Test di memoria . . . 69
6.1.1 Occupazione Nodo . . . 69
6.1.2 Occupazione libreria . . . 70
6.1.3 Occupazione funzioni di codifica e decodifica . . . 73
6.2 Prestazioni . . . 76
6.3 Performance EXI . . . 79
6.4 Conclusioni . . . 83

A Schemi XSD 85
A.1 Schemi progetto SENSEI . . . 85

B Documenti XML 89
B.1 Documenti XML SENSEI . . . 89

Bibliografia **90**

Sitografia **91**

Elenco delle tabelle **95**

Elenco delle figure **96**

Elenco dei listati **99**

Acronyms

DRY Don't Repeat Yourself

SAX Simple Api for XML

UML Unified Modeling Language

W3C World Wide Web Consortium

WSN Wireless Sensors Network

SENSEI Integrating the Physical with the Digital World of the Network of the Future

WS&AN Wireless Sensor and Actuator Networks

API Application Programming Interface

XML eXtensible Markup Language

EXI Efficient XML Interchange

XSD XML schema definition

Capitolo 1

Introduzione

1.1 Scenario

La costante evoluzione tecnologica porta sempre di più alla diffusione nella vita di tutti i giorni di dispositivi elettronici in grado di interconnettersi tra loro.

Negli ultimi anni la gamma di dispositivi in grado di interconnetersi mediante la rete *internet* si sta progressivamente allargando. I *personal computer* sono sempre più affiancati da *tablet pc* e *smart phone*, i consumi energetici acquistano sempre più importanza dato che questi apparecchi vengono principalmente usati a batteria.

Parallelamente si stanno diffondendo altre categorie di dispositivi elettronici detti ***sensori*** con *hardware* molto più limitato e non destinati al mercato *consumer*, ma in grado di comunicare tra loro mediante canale radio e di interagire con l'ambiente che li circonda mediante periferiche di rilevazione dati (temperature, luce, umidità ma non solo) e mediante periferiche di output in grado di inviare segnali elettrici ai dispositivi a loro eventualmente collegati.

La rete diventa sempre più il mezzo tramite il quale comunicano le applicazioni che diventano orientate al *web*. Le informazioni scambiate sono costituite sia da dati che da comandi di controllo remoto. E questi applicazione stanno progressivamente arrivando a tutte le tipologie di dispositivi di cui abbiamo finora parlato.

1.2 XML e EXI

In questo scenario l'esigenza di un metodo comune di codificare lo scambio delle informazioni ha dato vita a eXtensible Markup Language (XML).

Nato come formato di interscambio per il web, acquisisce progressivamente sempre maggiore importanza. Le sue caratteristiche di estensibilità, versatilità ed espressività lo rendono infatti il formato ideale per il dialogo tra le applicazioni.

D'altra parte, l'ingresso in questo scenario di dispositivi con risorse limitate, dal punto di vista di memoria, capacità di calcolo, autonomia e banda di comunicazione mette in luce i limiti intrinsechi di XML. Il costo di processamento e la verbosità del linguaggio lo rendono infatti poco adatto all'utilizzo su dispositivi integrati.

A partire dal *Binary Interchange Workshop* nel Settembre 2003 e proseguendo con l'attività del *XML Binary Characterization Working Group* in seguito fondato, il World Wide Web Consortium (W3C), il consorzio che si occupa della standardizzazione del *web* in tutte le sue forme, ha cominciato a studiare queste problematiche. I risultati di questo gruppo di lavoro hanno portato alla definizione dei requisiti che una codifica alternativa del formato XML avrebbe dovuto rispettare. Sucessivamente si è creato il *The Efficient XML Interchange Working Group*, ancora attivo, che ha prodotto e valutato varie proposte di formato, in seguito ha selezionato Efficient XML Interchange (EXI) e si occupa ora della sua definizione e standardizzazione.

Il W3C sta quindi definendo (attualmente allo stato di *candidate reccomendation*) una specifica, disponibile in [9], per un formato chiamato EXI, che si propone di essere una rappresentazione estremamente compatta delle informazioni esprimibili con XML. EXI si propone di ottimizzare contemporaneamente sia le prestazioni sia l'uso delle risorse *hardware*, per essere appunto particolarmente adatto all'impiego su dispositivi mobili ed in generale per ridurre il volume dei dati scambiati tra le applicazioni.

Come si vede in figura 1.1 EXI e XML sono pensati come il mezzo di collegamento tra dispositivi elettronici eterogenei.

Una trattazione più ampia di questi argomenti viene rimandata al capitolo 4

1.3 SENSEI

Nello stesso scenario si sviluppa il progetto europeo denominato Integrating the Physical with the Digital World of the Network of the Future (SENSEI). La *vision* di questo progetto è quella di un ambiente intelligente formato da un insieme eterogeneo di reti di sensori e attuatori wireless (Wireless Sensor and Actuator Networks (WS&AN)) i cui servizi possono essere resi disponibili mediante un framework utilizzabile su larga scala accessibile mediante un'interfaccia universale.

Figura 1.1: Uso di EXI e XML per le comunicazioni in reti di dispositivi eterogenei

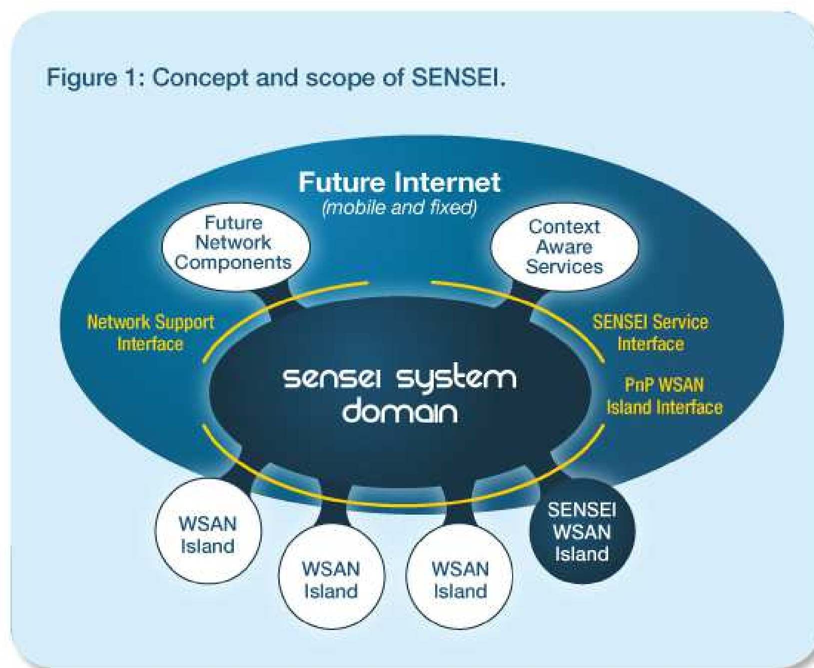

Figura 1.2: Schema del progetto SENSEI

Come rappresentato in figura 1.2 il progetto SENSEI mira a fondere in un unico sistema tutte le reti di servizi mobili o fisse collegate ad Internet creando un unico grande dominio. All'interno di questo vasto scenario trova

collocazione lo sviluppo di un servizio che permetta di comunicare da remoto con ogni singolo dispositivo che compone la rete di sensori in modo da acquisire dati e informazioni ambientali. Un sistema di questo tipo è rappresentato in figura 1.3 in cui vi sono una o più reti di sensori dislocate in qualche parte del mondo e sono connesse alla rete Internet mediante un punto di accesso denominato WSN Gateway che è collegato ad un Application Server. I dispositivi della rete di sensori sono accessibili mediante IPv6, implementato grazie al protocollo 6LoWPAN, mentre la rete Internet che connette il WSN Gateway e l'Application Server utilizza il protocollo IPv4 e solo in futuro passerà alla nuova versione.

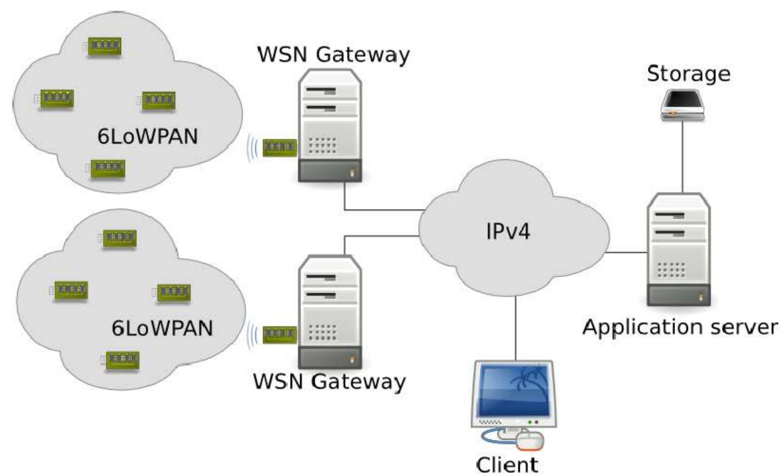

Figura 1.3: Scenario globale del sistema

Uno dei compiti del gateway è quello di garantire l'interoperabilità delle due versioni del protocollo. Nell'Application Server è eseguita un'applicazione web che permette quindi di essere raggiungibile da un browser Internet utilizzabile da qualsiasi dispositivo che lo supporti e di interagire attraverso il gateway con la rete di sensori.

1.4 Contributo dell'autore

All'interno del progetto SENSEI si è quindi manifestata la necessità di stabilire quale formato per lo scambio dei dati tra i sensori e le applicazioni utilizzare ed è stato scelto XML. Tuttavia a causa della natura dei sensori

le loro limitazioni *hardware* impediscono l'implementazione di un processore XML al loro interno.

Si è quindi scelto di sviluppare una implementazione dello standard EXI che fosse utilizzabile nei sensori *wireless* impiegati nel progetto e più in generale su dispositivi con risorse limitate che necessitino di una implementazione di EXI.

In questo lavoro di tesi ci si è quindi occupati della programmazione di una libreria *software* che implementi le funzionalità essenziali dello standard EXI e che sia efficientemente scalabile da essere utilizzabile sia sui nodi sensori che su un normale computer.

Capitolo 2

Piattaforma Hardware

Gli sviluppi della tecnologia wireless consentono, allo stato attuale, di produrre dispositivi di piccole dimensioni dotati di interfaccia radio a costi molto contenuti. Queste piattaforme sono chiamate ***sensori*** o ***nodi*** wireless e sono generalmente dotati di un microcontrollore che ne gestisce il funzionamento. Possono fungere da dispositivi di rilevamento ambientale e attuatori per interagire con l'ambiente che li circonda.

La tecnologia radio di cui sono dotati li rende appetibili anche per la sostituzione di impianti cablati. Tuttavia una delle loro più grandi potenzialità sta nel fatto di poter creare una ***rete di sensori***.

2.1 Wireless Sensor Network

Una Wireless Sensors Network (WSN) o rete di sensori senza fili consiste di un insieme di nodi che possiedono le già indicate capacità di interazione con l'ambiente che li circonda. Sono in grado di comunicare tra di loro mediante l'intefaccia wireless di cui sono dotati ed hanno a disposizione risorse limitate sia dal punto di vista della memoria che dell'energia (sono solitamente alimentati a batteria), caratteristiche cruciali che limitano fortemente la programmazione di applicazioni.

Lo scopo principale di queste WSN è la raccolta di dati ambientali che avviene mediante i dispositivi di rilevazione dei quali sono equipaggiati i nodi: temperatura, umidità, luminosità, pressione ... Queste informazioni vengono solitamente inviate ad uno speciale nodo che nel progetto SENSEI è denominato *gateway* in genere collegato ad un computer che si occupa dell'elaborazione e aggregazione dei dati. Il sistema solitamente consente anche il processo di interrogazione: l'applicazione contatta il computer a cui

è collegato il *gateway* e quest'ultimo invia dei messaggi di richiesta dati ad uno o più nodi della WSN.

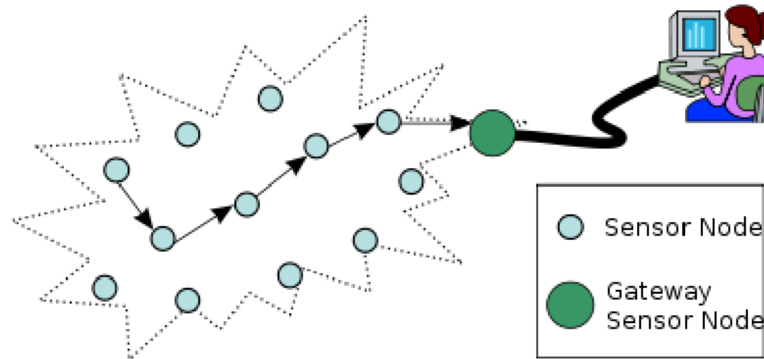

Figura 2.1: Wireless Sensors Networks

I nodi, grazie al microcontrollore programmabile che ne gestisce il funzionamento, possono anche effettuare semplici elaborazioni sui dati (come ad esempio semplici medie temporali) o operazioni più complesse come instradamento dei pacchetti all'interno della rete fino a codifiche di formato, come quella implementata nella libEXI oggetto di questa tesi.

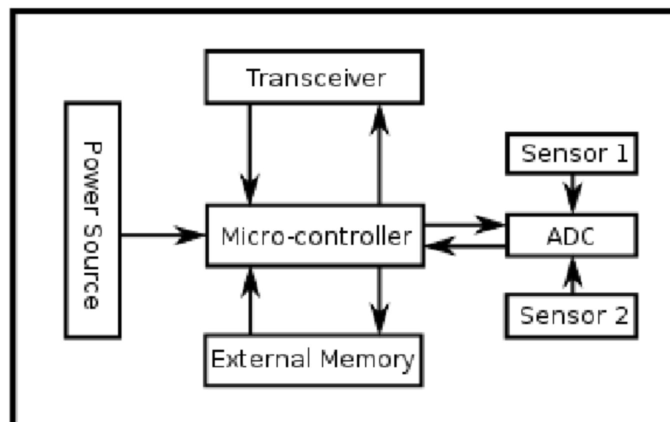

Figura 2.2: Architettura tiny node

Gli ambiti applicativi delle WSN sono i più vari e spaziano dall'ambito militare[1] a quello civile[2]. Per quanto riguarda il primo ambito, si può

semplicemente pensare ai nodi sensori utilizzati come parte integrante della strategia C4ISRT (controllo, comunicazione, calcolo, intelligence, sorveglianza, riconoscimento) verso la quale sta evolvendo l'esercito moderno. Le capacità di dislocamento rapido, l'alta tolleranza ad errori o alla perdita di alcuni dei nodi della rete tipiche delle reti di sensori li rendono preferibili (ed integrabili) alle postazioni di sensori tradizionali. Reti di sensori si potrebbero infatti usare in operazioni di monitoraggio dei campi di battaglia sia per il rilevamento dei movimenti che per la stima delle zone colpite da un attacco.

Anche in ambito civile l'utilità delle reti di sensori è molteplice e spazia dalla domotica (sia per il controllo remoto degli elettrodomestici che per il loro comportamento gestito in base ai dati raccolti dai sensori), al monitoraggio a distanza dei pazienti di un ospedale fino al rilevamento ambientale come nel caso di reti di sensori in zone ad alto rischio di incendi.

Esiste una gran varietà di piattaforme hardware su cui sviluppare una rete di sensori: quelle utilizzate in questo progetto sono stati i TelosB e i TMote Sky dell'università di Berkeley. Queste piattaforme hanno prestazioni pressochè identiche e verranno utilizzate senza distinzione nel seguito.

2.2 Telosb/TMote Sky

Queste piattaforme ([3] e [4]) sono state originariamente progettate all'università di Berkeley [13] e sono ora prodotte dalla Crossbow Technology [14]. Sono dotati di un microprocessore a 8 MHz della Texas Instruments, MSP 430, con 10 kB di RAM e 48 kB di flash per la memorizzazione dell'applicazione e di 1 MB di memoria flash esterna per l'archiviazione dei dati.

Utilizzano lo standard IEEE 802.15.4 a 2.4 GHz grazie al chip CC2420 che fornisce una velocità trasmissiva di 250 kbps, un basso consumo di energia e algoritmi di cifratura di tipo AES-128 a livello MAC. Sono forniti di un'antenna integrata che garantisce una copertura di 50m in ambienti chiusi e 125m in campo aperto. Vengono programmate mediante connettore USB che può essere utilizzato per scaricare i dati immagazzinati nella memoria flash esterna. La presa USB viene anche impiegata per alimentare il nodo oppure, in sostituzione, si possono utilizzare 2 batterie di tipo AA. Come affermato precedentemente queste piattaforme possono essere equipaggiate con diversi tipi di sensori: Hamamatsu S1087 e Hamamatsu S1087-01 per il monitoraggio della luce visibile e dell'infrarosso, Sensirion SHT11 per il controllo della temperatura e dell'umidità. Sono inoltre forniti di un connettore a 16 pin per estendere le funzionalità con ulteriori componenti hardware.

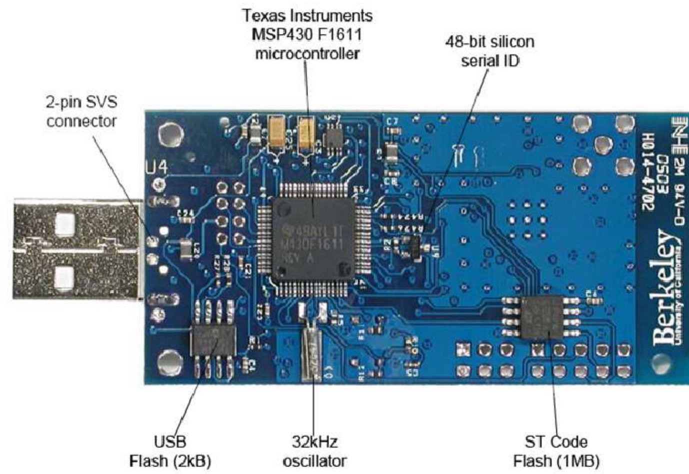

Figura 2.3: Il nodo TMote Sky visto dal basso

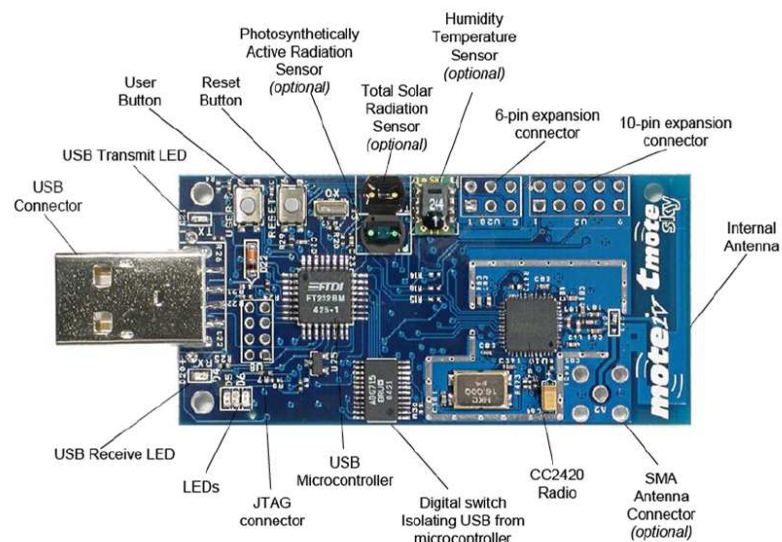

Figura 2.4: Il nodo TMote Sky visto dall'alto

Capitolo 3

Piattaforma Software

La libreria sviluppata è, come si vedrà in dettaglio nel capitolo 5, divisa in due parti che hanno ambiti di utilizzo differenti e che sono state implementate usando due diversi linguaggi di programmazione.

La prima parte della libreria, chiamata anche ***preprocessore*** è infatti pensata per essere eseguita su normali *PC* ed è stata scritta, data la complessa natura del problema che deve affrontare, nel linguaggio ad alto livello ***ruby*** [23].

La seconda parte della libreria (detta ***processore***) è invece pensata per essere eseguita sia sui normali *PC* che sui *nodi sensori* ed è stata scritta quindi in linguaggio ***C*** [5].

I nodi sensori usati nel progetto SENSEI in realtà usano un sistema operativo chiamato ***Tiny OS*** [16] scritto in una *variante* del *C* chiamato ***NesC*** che mantiene però la compatibilità con il codice *C* usato nello sviluppo della libEXI.

Infine, il linguaggio XML [24] è usato come formato di interscambio dati all'interno e all'esterno della rete di sensori, attorno ad esso si sviluppa la tesi.

3.1 Il linguaggio Ruby

È qui esposta una veloce introduzione al linguaggio utilizzato per la stesura della prima parte della libreria, per dare quindi al lettore la possibiltà di comprendere meglio perchè si è scelto di utilizzarlo. Si rimanda nuovamente al capitolo 5 per una trattazione esaustiva del codice sviluppato.

Le caratteristiche principali che hanno portato alla scelta di *Ruby* [23] sono il suo essere un linguaggio ad alto livello, fortemente orientato agli oggetti e con una sintassi semplice ma molto espressiva. Queste peculiarità

hanno molto avantaggiato lo sviluppo del preprocessore dato che il suo scopo è principalmente l'elaborazione di documenti XML e la creazione di oggetti complessi a partire dall'informazione in essi contenuta.

3.1.1 Cenni storici

Ruby è stato creato da Yukihiro Matsumoto (Matz) in Giappone. Egli si rese conto che uno dei maggiori problemi coi quali si scontra un programmatore è l'enorme impiego di tempo per la progettazione, lo sviluppo e la manutenzione del codice inoltre spesso non è richiesta una soluzione estremamente performante, con questa idea fissa nella mente egli ideò un linguaggio di programmazione semplice da usare e da apprendere e con un'elevata affidabilità anche per grossi progetti, ma soprattutto che si concentrasse sull'uomo più che sulla macchina.

Per la creazione di Ruby Matz si è ispirato prevalentemente a Smalltalk, un linguaggio di programmazione orientato agli oggetti sviluppato allo Xerox PARC prevalentemente per un uso didattico e che ha pesantemente influenzato altri linguaggi come C e Java. Ruby permette applicazioni molto più varie di quanto si possa pensare, è semplice da estendere, relativamente leggero e con una portabilità davvero elevata. Come già detto Ruby è stato progettato soprattutto per questi tipi di problemi:

- Text processing: classi specifiche rendono efficiente l'elaborazione di testi;

- CGI programming: Ruby fornisce tutto ciò che serve per la programmazione di CGI (Common Gateway Interface), incluse classi per la manipolazione di testi, una libreria CGI, interfacce per database, e anche eRuby (embedded Ruby) un `mod_ruby` per Apache;

- Network programming: la programmazione di reti può avvalersi di classi specificamente strutturate in Ruby;

- GUI programming: sono disponibili alcuni ambienti di sviluppo con interfaccia grafica come ad esempio Ruby/Tk e Ruby/Gtk;

- XML programming: Ruby, grazie alle elevate potenzialità di elaborazione dei testi e potendosi avvalere di UFT-8 (Unicode Transformation Format, 8 bit) rende la programmazione in XML davvero semplice. Inoltre è disponibile un'interfaccia alla libreria ***libxml2***, una delle più popolari librerie di processamento XML ed è anche quella usata in questa tesi;

- Prototyping: grazie alla sua alta produttività Ruby e spesso usato per sperimentare nuove soluzioni.

3.1.2 I Principi di base Ruby

Per un programmatore una delle cose più divertenti è potersi concentrare sul lato creativo dei propri programmi, Ruby tiene molto in considerazione questo aspetto. Fondamentalmente un linguaggio di programmazione non è che un interfaccia per l'utente, di conseguenza è conveniente che segua i seguenti principi:

Principio di Sinteticità
I computer sono i propri servi, non i propri padroni. Quindi bisogna poter dal loro ordini velocemente, un buon servo deve fare molto lavoro con un ordine breve.

Principio di Consistenza
Ruby è un linguaggio relativamente semplice, ma non troppo semplice, è stato pensato in modo da agevolare chi proviene da altri linguaggi di programmazione, quindi la sua unicità non è troppo marcata e lo rende di facile apprendimento.

Principio di Flessibilità
Le lingue servono per esprimere il pensiero, una lingua non dovrebbe restringere il pensiero umano, anzi dovrebbe aiutarlo. Ruby consiste di un piccolo nucleo fisso (che è la sua sintassi) e di una serie di librerie liberamente estendibili. Poiché la maggior parte delle cose è fatta con le librerie, è possibile trattare le classi definite da un utente e gli oggetti esattamente come quelli propri del linguaggio. Grazie a questi principi la programmazione in Ruby è decisamente meno stressante.

3.2 TinyOS, C e NesC

Tradizionalmente la programmazione di microcontrollori non richiede l'utilizzo di sistemi operativi, il programmatore infatti si avvale solitamente dell'astrazione offerta da un linguaggio di programmazione e scrive il suo codice che è in grado di interfacciarsi con l'*hardware*, viene compilato (tradotto in linguaggio macchina) e caricato sul microcontrollore per essere eseguito.

Il linguaggio ***C***[5], sia per motivi storici che per le sue caratteristiche è uno dei più usati nella programmazione di microcontrollori. Il *C* resta, anche a distanza di anni dalla sua creazione, probabilmente il linguaggio più performante e quello che ha il miglior rapporto tra possibilità astrattive e capacità di controllo sui più piccoli dettagli dell'*hardware*.

In un'implementazione su larga scala, come quella che coinvolge una WS&AN, bisogna però considerare che la rete, essendo eterogenea per natura, coinvolge dispositivi tra loro diversi, e anche gli stessi nodi sensori possono montare hardware differente. Inoltre, molti sviluppatori sono coinvolti in un progetto come SENSEI ed è quindi importante l'interoperabilità del codice.

Per questi motivi il codice sviluppato nell'ambito del progetto SENSEI si avvale di un ulteriore strato di astrazione, viene infatti usato un ***sistema operativo*** per nodi sensori che consente al codice prodotto di lavorare su diverse piattaforme *hardware*.

I sistemi operativi (OS) implementati per le reti di sensori sono molto diversi rispetto a quelli conosciuti e utilizzati dagli utenti di un qualsiasi PC: la differenza fondamentale è dovuta alla mancanza di uno strumento che consenta l'interfacciamento in tempo reale tra l'utente e il nodo.

Questi sistemi devono garantire flessibilità e modularità per un loro utilizzo su diverse piattaforme, un utilizzo efficiente delle risorse per ottimizzare il consumo energetico e, a causa della limitata memoria di cui sono forniti i tiny node, non devono essere complicati per quanto concerne la loro programmazione.

Esistono differenti OS per queste piattaforme hardware come: Contiki [15], BTnut[17], MANTIS [18], Nano-RK [19], SOS [20] ma quello utilizzato in questo progetto è stato ***TinyOS*** [16] che a differenza dei precedenti non è scritto nel linguaggio *C* ma nella sua variante *NesC*.

3.2.1 NesC

NesC è un linguaggio di programmazione simile al C, progettato appositamente per sviluppare TinyOS. Il concetto base di NesC sono i componenti o moduli che vengono assemblati o connessi per costituire l'intero programma.

In ogni componente sono definite due entità: un elenco di nomi di interfacce che possono essere usate (*uses*) o fornite (*provides*) e la loro implementazione. Nell'interfaccia, come mostra la figura 3.1, sono definiti comandi (*command*) che sono implementati dai componenti che provvedono quell'interfaccia e gli eventi (*event*) che sono implementati dai moduli che utilizzano quell'interfaccia.

Questo permette di generare una complessa connessione bidirezionale tra i differenti componenti che costituiscono l'intero programma: il modulo che utilizzerà un'interfaccia esegue una chiamata (*call*) ad un comando che sarà implementato nell'altro componente (ad esempio *send*) ma dovrà descrivere il comportamento dell'evento segnalato (*signal*) al termine di quella particolare chiamata (ad esempio *sendDone*). I comandi sono non bloccanti e il loro completamento è segnalato da un evento.

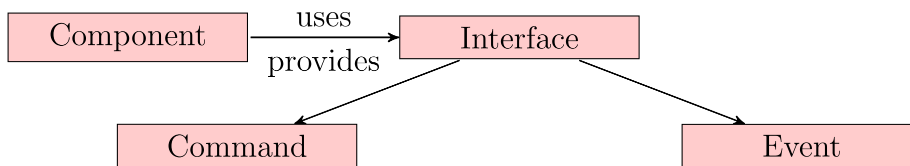

Figura 3.1: Componenti e Interfacce in NesC

Di solito la direzione della chiamata di un comando avviene verso il basso, cioè da un'applicazione a più alto livello ad un componente più vicino al livello *hardware*, mentre la segnalazione di un evento ha senso opposto.

Per ogni singolo componente devono essere scritti due file: il primo che descrive l'elenco delle interfacce e la loro implementazione ed è identificato dalla lettera *P* dopo il nome e un secondo denominato file di configurazione, riconosciuto tramite la lettera *C*, che viene utilizzato dal sistema operativo per connettere i differenti moduli tra di loro sulla base delle interfacce che vengono utilizzate.

Questa connessione tra i moduli viene effettuata in maniera statica per risparmiare energia e memoria limitando però la versatilità: devono essere specificati tutti i collegamenti e non è possibile farlo in una fase successiva. Anche l'allocazione di memoria è statica e devono essere dichiarate le dimensioni degli array in fase di scrittura del codice.

3.2.2 C

La parte di codice detta ***processore*** è stata quindi sviluppata in linguaggio *C*, l'insieme di funzioni usate si rifà quasi esclusivamente a quelle fornite dal linguaggio standard nella sua versione ANSI C 99 [5].

Per questo il codice prodotto può essere compilato su architetture a `16, 32 e 64 bit` semplicemente cambiando il compilatore ed i suoi parametri.

Dato che il *NesC* è una variante del *C* la libreria può essere inclusa senza problemi all'interno del codice del progetto SENSEI.

3.2.3 TinyOs

TinyOS è il primo sistema operativo pensato e realizzato per una rete di sensori. Questo OS è open source ed è basato sul paradigma di programmazione *event-driven*, dove cioè il flusso del programma è determinato dal susseguirsi di eventi, come la pressione di un pulsante o la ricezione di un pacchetto e il

compito del programmatore è di specificarne il comportamento tramite una sequenza di istruzioni.

Il *kernel* gestisce, oltre agli eventi, anche i *task* e la concorrenza tra i due viene gestita nel seguente modo: gli eventi possono prendere possesso del processore, sono cioè di tipo *preemptive*, anche se in quel momento sono in esecuzione altri eventi o *task*; i *task* non hanno diritto di prelazione (*non-preemptive*) su altri eventi e *task*, e viene utilizzata una coda FIFO per mantenere l'ordinamento dei task.

La caratteristica più evidente di TinyOS è quella di fornire un'architettura *component-based*, dove è possibile il riutilizzo di più componenti che rappresentano un'astrazione dell'implementazione *hardware*, come ad esempio gli stessi sensori di cui il tiny node può essere equipaggiato, l'interfaccia per il controllo del timer, il sistema per il controllo del dispositivo radio, strumenti per l'archiviazione dei dati in memoria. La maggior parte di questi componenti sono inclusi nella libreria del sistema operativo e vengono connessi tra di loro per poter essere utilizzati.

3.3 Il codice del progetto SENSEI - Scenario Applicativo

Con riferimento a quanto detto nel capitolo introduttivo si esamina ora con maggior dettaglio lo scenario di riferimento di questa tesi.

La libreria prodotta infatti pur essendo portabile e slegata dal progetto SENSEI ha certamente preso il suo scenario applicativo e il suo attuale stato di sviluppo come riferimento principale.

In figura 3.2 vediamo nuovamente riproposta l'architettura della rete SENSEI.

Gli agenti coinvolti nel lavoro di tesi sono:

- il *client*;
- il *WSN Gateway*;
- i nodi sensori.

Uno degli obbiettivi del progetto SENSEI è infatti quello di rendere ogni nodo di una WSN raggiungibile singolarmente da un *client* connesso alla rete *internet*.

A questo scopo lo sviluppo di SENSEI ha portato alla scelta di superare il modulo di comunicazione standard di *TinyOS* ed attribuire ad ogni nodo un indirizzo IPv6 [25] in modo da poter indirizzare ogni nodo univocamente.

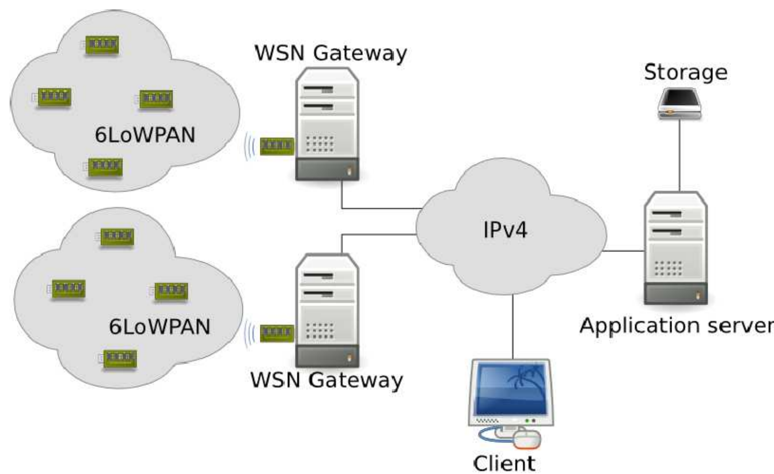

Figura 3.2: Scenario globale del sistema

L'implementazione fa uso di *6lowPan* e si rimanda alla lettura di [6] e [7] per una dettagliata spiegazione.

Sempre con riferimento a [6] si vede ora lo stack protocollare a 5 livelli, sviluppato su *TinyOS* e sul quale poggia la libEXI.

- il primo e più alto livello si occupa del comportamento dei sensori di cui il nodo è equipaggiato;
- il secondo contiene il sistema per consentire l'accesso alle risorse del livello sovrastante mediante un interfaccia *Resource Access* o *RA* e un modulo che consenta di pubblicare, *Resource Publication* o *RP*, il descrittore che contiene le specifiche del nodo stesso;
- nel terzo livello trova collocazione il *Binary Web Service* o *BWS* che è un middleware che offre un servizio di tipo Client/Server alle differenti applicazioni del livello superiore;
- il quarto ospita i protocolli di comunicazione *UDP* e *6LoWPAN*;
- nel quinto e ultimo livello si trova lo standard di rete IEEE 802.15.4.

Uno dei componenti fondamentali di questa architettura è quindi il *Binary Web Service* [6], un sistema software progettato per portare l'interoperabilità di differenti dispositivi interconnessi in una rete. In altre parole i nodi sono equipaggiati con un server Web minimale in grado di accettare richieste

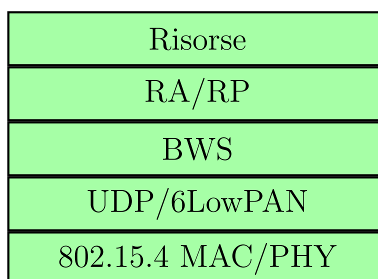

Figura 3.3: Architettura a livelli del sistema

HTTP di tipo PUT, GET, POST e DELETE, di interpretarle e di passarle ai livelli superiori.

L'utente che conosca l'indirizzo IPv6 del nodo può quindi effettuare una richiesta di GET sul valore di un certo sensore del nodo, la richiesta viene raccolta dal Web Server di cui è dotato il *gateway*, tradotta nel formato 6LowPan e ricevuta dal componente BWS del nodo.

La richiesta viene poi trasferita ai componenti superiori fino ad arrivare a quello che si occupa della gestione delle risorse, il dato viene letto ed è a questo punto che sarebbe necessario usare XML.

Inoltre il nodo deve poter effettuare il procedimento inverso, accettare cioè richieste complesse, che gli vengono espresse in formato XML interpretarle ed agire di conseguenza.

Il nodo infatti deve quindi interpretare e rispondere le richieste che riceve, ma i requisiti di interoperabilità sopra citati, alla base del progetto SENSEI, richiedono che la risposta sia in un formato di scambio dati condiviso.

Il formato scelto per l'interscambio dei dati tra i nodi ed il *gateway* è XML che non può però essere usato direttamente nelle reti di sensori. In particolare XML è prolisso e quindi la trasmissione di un intero documento può essere decisamente costosa sia in termini di banda che di risorse energetiche e di memorizzazione.

Per questi motivi (spiegati in dettaglio al capitolo 5) si è scelto di implementare un codificatore EXI che consentisse di passare da XML al suo formato compresso (EXI appunto) e viceversa.

EXI consente infatti di codificare l'informazione contenuta in un documento XML in maniera particolarmente efficiente, mantenendo la struttura del documento, ed anzi utilizzando l'informazione sulla stessa per ottimizzare il risultato della codifica.

Si ricorda nuovamente che la libreria sviluppata non è vincolata al codice contenuto in *TinyOS* o al codice sviluppato per il progetto SENSEI

anche se il sottoinsieme di funzioni previste dalla specifica EXI scelte per questa implementazione certamente risente dei requisiti che emergono dal su sviluppo.

Capitolo 4

Lo standard EXI

Nelle sezioni precedenti si è visto come l' XML sia il linguaggio di formattazione più versatile tra quelli comunemente usati, e per questo il suo impiego sta negli anni uscendo dall'ambito del Web per arrivare ad essere lo standard *de Facto* nell'interscambio di dati tra applicazioni diverse. XML è ad esempio alla base dello standard Open Document (ODF) utilizzato da molte suite di programmi per ufficio (da OpenOffice a Google Docs).

Il limite maggiore all'impiego di XML su device che dispongono di risorse limitate, come nel caso dei sensori senza fili, sta nel fatto che in questo linguaggio il rapporto tra parole di codice e contenuto informativo è molto elevato. In altri termini, molti dei byte di un documento XML servono a definire la struttura del documento più che a veicolare l'informazione in esso contenuta.

Da queste e ed altre riflessioni nasce l'idea del W3C di definire uno standard più efficiente, ovvero EXI[1].

Come espresso nella *Candidate Raccomandation* (la versione attuale dello standard EXI, al momento in fase finale di definizione) [9] le principali caratteristiche di EXI sono:

- EXI è una rappresentazione estremamente compatta di XML;
- EXI intende ottimizzare le prestazioni e l'utilizzo delle risorse computazionali;
- EXI usa un approccio ibrido che sfrutta sia le teorie dei linguaggi che tecniche di compressione empiriche, verificate dalle misurazioni;
- EXI usa pochi tipi di dato, ma senza perdita di informazione;

[1]Il W3C è il consorzio internazionale che si occupa di definire tutti gli standard che riguardano il Web

- L'algoritmo alla base di EXI è ideale per implementazioni veloci e compatte.

EXI è quindi uno standard proposto dal W3C per la creazione di un formato di interscambio che consista in una versione compressa del noto XML, utile in scenari nei quali sia considerato importante l'aspetto relativo al volume dei dati scambiati.

Questa compressione viene peró raggiunta sfruttando al massimo l'informazione sulla struttura del documento XML da codificare. Essa può essere dedotta dal documento durante l'operazione di codifica o può essere estrapolata da documenti di tipo XML schema definition (XSD) o equivalenti[2].

Lo standard proposto in [9] definisce quindi i seguenti elementi di EXI:

- la struttura del pacchetto EXI;
- i tipi di dato supportati dallo standard;
- le operazioni di Codifica e Decodifica dei tipi di dato;
- le modalità di Generazione delle grammatiche di un documento XML;
- le operazioni di Codifica e Decodifica delle suddette grammatiche;
- le opzioni che il codificatore EXI deve supportare;
- ulteriori tecniche di compressione euristiche.

Rimandando a [9] per una trattazione esaustiva dello standard si cercherà in questa sede di spiegare in breve i principi che stanno alla base della codifica EXI, per consentire una migliore comprensione del lavoro di tesi.

Nel resto del capitolo verranno descritte le parti dello standard più utili alla comprensione del lavoro di implementazione effettuato. Tale lavoro si è infatti occupato maggiormente di alcune parti dello standard come spiegato in dettaglio al capitolo 5.

Si sottolinea però che l'implementazione ottenuta è tale da essere compatibile con altre implementazioni dello standard.

4.1 XML e grammatiche

Alla base della codifica EXI sta il concetto di ***grammatica***

[2]Documenti di questo tipo contengono infatti descrizioni di struttura. Definiscono quindi una classe di documenti XML con caratteristiche comuni

Con grammatica si intende una funzione che mappa tra loro due insiemi di simboli (detti simboli terminali e simboli non terminali) appartenenti ad un alfabeto.

Più formalmente si definisce grammatica una quadrupla $G = \langle N, \Sigma, P, S \rangle$:

1. N sono i simboli non terminali;
2. Σ sono i simboli terminali;
3. $S \in N$ detto assioma è il simbolo non terminale di inizio;
4. P Sono le regole di produzione.

Il linguaggio generato (o riconosciuto) da una grammatica formale G denotato come $L(G)$, viene definito come tutte quelle stringhe su Σ che possono essere generate iniziando con l'assioma S e poi applicando iterativamente le regole produttive di P fino a quando non vi siano più simboli non terminali.

Un documento XML è caratterizzato da un insieme di *grammatiche regolari*, grammatiche a cui corrispondono quindi *automi a stati finiti*[3] che sono in grado di riconoscere lo stesso linguaggio.

In una *grammatica regolare* ogni produzione ha due parti, e destra. Nella parte sinistra c'è sempre un simbolo non terminale mentre la parte destra può contenere un terminale seguito da un terminale oppure un singolo terminale.

Una grammatica assume quindi la forma:

$$A \rightarrow \alpha B$$
$$B \rightarrow \beta$$

con A, B simboli non terminali e β, γ simboli terminali e A assioma o simbolo di inizio.

L'automa associato ̀presentato in figura 4.1.

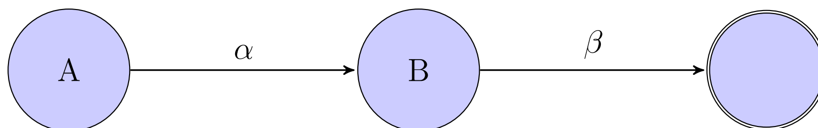

Figura 4.1: Automa associato ad una semplice grammatica regolare

XML viene solitamente definito *metalinguaggio*, questo per indicare che il suo scopo è la definizione di altri linguaggi artificiali detti *linguaggi obbiettivo*.

[3]Si definisce automa a stati finiti (FSA) . . . XXX

Mediante XML si creano infatti nuovi linguaggi atti a descrivere informazione strutturata.

La definizione di un documento XML implica quindi la definizione del suo *schema*, della sua struttura. È necessario cioè stabilire come deve essere espressa l'informazione.

Questa operazione è analoga alla definizione di un insieme di *grammatiche regolari* che stabiliscono appunto quale sia la struttura valida di un documento.

4.2 Lo Stream EXI

Un flusso (stream) EXI è costituito da una intestazione (header) seguita da un corpo (body), si veda a proposito la figura 4.2.

L'*header* del pacchetto contiene le informazioni di versione sul formato, e può opzionalmente veicolare le informazioni sulle opzioni di codifica usate per il corpo EXI. Se questa parte non è presente (come nel caso della nostra implementazione) si assume che queste informazioni siano passate *out of band*[4].

Il corpo del pacchetto contiene invece una sequenza di blocchi elementari detti *eventi* che descrivono il documento codificato.

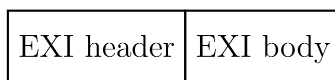

Figura 4.2: struttura pacchetto EXI

Nella tabella 4.1 si può trovare una descrizione delle opzioni principali dell'intestazione EXI

Una nota particolare va fatta per l'opzione *alignement* che può assumere due valori: nel caso del valore *bit packed* ogni elemento dello stream viene codificato usando il minor numero di *bit* possibile, nel caso del valore *byte packed* invece si usa il numero minimo di *byte*. Il vantaggio che la prima da sulla seconda in termini di dimensione dello *stream* risultante ha, come vedremo in seguito, una controparte in una più difficile scrittura e gestione del codice del codificatore EXI, motivo che ha portato il codice sviluppato in questa tesi ad implementare solamente la seconda modalità.

[4]Con questo termine si intende che il passaggio di una certa informazione avviene mediante un generico canale di comunicazione non definito nella specifica la cui gestione viene lasciata a chi si occupa dell'implementazione.

Tabella 4.1: opzioni header EXI

EXI Option	Description	Default Value
alignment	Alignment of event codes and content items	bit-packed
compression	Indicates if EXI compression is to be used for better compactness	false
strict	Strict interpretation of schema is used to achieve better compactness	false
fragment	Indicates if the body is to be encoded as an EXI fragment instead of an EXI document	false
preserve	A set of options that controls whether comments, processing instructions, etc. are preserved	all false
selfContained	Enables self-contained elements. Self-contained elements may be read independently from the rest of the EXI body	false
schemaID	Identifies the schema used during encoding	*no default value*
datatypeRepresentationMap	Identify datatype representations used to encode values in EXI body	*no default value*
blockSize	Specifies the number of Attribute (AT) and Character (CH) values for each block used for EXI compression	1,000,000
valueMaxLength	Specifies the maximum string length of value content items to be considered for addition to the string table	*unbounded*
valuePartitionCapacity	Specifies the total capacity of value partitions in a string table	*unbounded*
user defined meta-data	User defined options may be added	*no default value*

Tabella 4.2: Eventi EXI supportati

EXI Event Type	Grammar Notation	Content
Start Document	SD	
End Document	ED	
Start Element	SE(qname)	
End Element	EE	
Attribute	AT(qname)	value
Characters	CH	value

4.3 EXI come flusso di eventi

Evento:

I blocchi che costituiscono un corpo EXI sono gli Eventi. Un corpo EXI è infatti una sequenza di eventi che rappresentano un documento EXI.

Il concetto di evento è similare a quello che si può trovare nelle Simple Api for XML (SAX). Immaginiamo per esempio di leggere, riga per riga, un generico documento XML. Ogni elemento da noi incontrato, sia esso parte della struttura del documento, o sia esso un pezzo dell'informazione che il documento vuole veicolare, è un evento.

Si considerano eventi l'inizio del documento XML e la sua fine, l'inizio di un elemento o di un attributo, così come l'inizio del contenuto di un elemento.

Ogni elemento XMLè quindi codificato in uno o più eventi EXI; ad esempio, un elemento pluto che contenga un attributo pippo e un attributo paperino
`<pluto pippo=4 paperino=3/>`
viene tradotto in una sequenza di eventi del tipo
`SE(‘‘pluto’’) - AT(‘‘pippo’’) - AT(‘‘paperino’’)- EE`.
La specifica EXI [9] definisce degli acronimi con cui indicare questi eventi: ad esempio SE = Start Element, AT = Attribute, EE = End Element.

In tabella 4.2 si possono vedere i vari tipi di eventi EXI che sono stati considerati per questa tesi, per una lista completa degli eventi EXI si rimanda nuovamente a [9].

Nel flusso EXI gli eventi *attribute* e *character* sono gli unici che contengono contenuto informativo. Gli altri eventi sono invece relativi alla struttura del documento codificato.

4.4 Eventi e grammatiche

Gli eventi EXI che è possibile incontrare ad un certo punto di uno stream EXI sono determinati dalla *grammatica EXI* che caratterizza il documento di cui si sta effettuando la codifica. Come nei documenti XML gli eventi possono essere annidati ma non possono intersecarsi, devono cioè essere contenuti completamente l'uno nell'altro. Le grammatiche EXI riflettono l'informazione già contenuta nella grammatica XML del documento di cui si sta effettuando la codifica. La sezione 4.7 spiega con maggior dettaglio il procedimento di generazione delle grammatiche EXI.

Il processo di codifica di un documento XML in formato EXI avviene mediante la trascrizione degli eventi che lo caratterizzano. Si noti che in un dato punto del documento XML non si può verificare uno qualunque degli eventi possibili, ma solo un loro sottoinsieme in base alla struttura del documento stesso.

La nozione alla base di EXI è che la struttura di un documento XML può essere rappresentata come ***un'insieme di automi a stati finiti*** e la sequenza di eventi che viene codificata nel corpo EXI non è altro che la sequenza degli stati assunti dagli automi che descrivono la struttura del documento.

Dalla teoria del linguaggi sappiamo che ogni automa esprime un *linguaggio regolare* caratterizzato da una certa *grammatica regolare*.

Il codificatore EXI deve quindi, a partire dall'informazione contenuta nel documento XML e negli eventuali schemi XSD che lo caratterizzano, costruire le grammatiche che lo definiscono per poter, durante il processo di codifica, stabilire gli stati che ogni automa assume.

Le grammatiche definite dallo standard EXI sono chiamate *EXI grammars*, sono grammatiche regolari strutturate nella forma:

Listato 4.1: Esempio di grammatica EXI

```
Exi_grammar_1

        LeftHandSide 1 :
                Terminal 1    LeftHandSide 1
                Terminal 2    LeftHandSide 2
        LeftHandSide 2 :
                Terminal 1    LeftHandSide 1
                Terminal 2
```

Continuando il semplice esempio introdotto alla sezione precedente, una possibile grammatiche per l'elemento `pluto` potrebbe essere:

Listato 4.2: Esempio di grammatica per l'elemento pluto

```
Exi_grammar_pluto

        Attribute-pippo :
                AT("pippo")     Attribute-paperino

        Attribute-paperino :
                AT("paperino")     End-Element

        End-Element :
                EE
```

Ogni grammatica è composta da *produzioni* e ogni produzione al suo interno è composta da una parte destra (*left hand side*) e da una o più parti sinistre (*right hand side*).

Ricordando che ogni grammatica ha un automa a stati finiti associato si può immaginare che la parte destra delle produzioni siano gli stati dell'automa mentre le parti sinistre siano gli archi che portano ad altri stati.

La parte sinistra di una produzione è inoltre composta da 1 o 2 elementi, è obbligatoriamente presente un *simbolo terminale* (un evento EXI) ed opzionalmente può essere presente un *simbolo non terminale* che corrisponde alla parte destra di una produzione o detto altrimenti, allo stato di destinazione nell'automa.

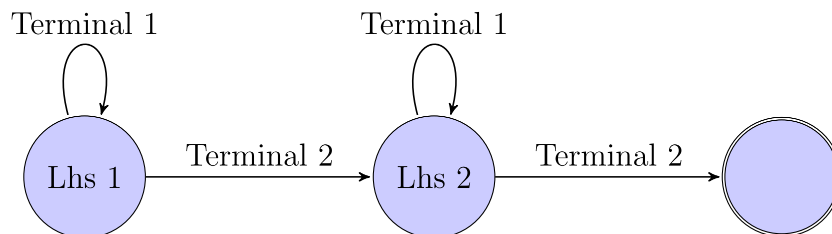

Figura 4.3: Grammatiche come automi

Nello stream EXI ogni evento non viene rappresentato con il suo nome, ma con un codice chiamato *Event Code*:

Si definisce event code una sequenza da 1 a 3 interi positivi che identificano un evento nello stream EXI.

Ogni parte destra di una produzione nella grammatica EXI ha infatti un *event code* associato, ovvero un valore che approssima la possibilità che un certo evento si verifichi, in questo modo eventi più probabili avranno *event*

code più bassi e quindi potenzialmente codificabili in meno *bit*. Queste probabilità sono state determinate in modo empirico e sono definite all'interno della specifica EXI: ad esempio si stabilisce che l'event code da associare ad un evento *Start Element* è sempre più basso di quello associato ad un evento *End Element*.

Le parti destre di ogni produzione sono infatti ordinate con una procedura, definita nello standard EXI, che mira a ordinarle in ordine probabilità decrescente. Gli *event code* sono invece assegnati in ordine crescente. *Event code* bassi esprimono quindi probabilità maggiori.

4.5 Il processo di codifica

Questa sezione illustra i passi del processo di codifica di un documento XML.

Le regole per la codifica di un documento EXI sono relativamente semplici, la codifica è guidata da un insieme di grammatiche che descrivono la struttura del documento XML e la conseguente struttura dello stream EXI.

Per comprendere meglio come avvenga ricordiamo un'altra definizione di EXI riportata in [10]:

> *EXI è una codifica informata che usa un insieme di grammatiche per determinare quali sono gli eventi che hanno maggior probabilità di accadere ad un certo punto di un flusso EXI e codifica le alternative più probabili in meno bit.*

Il codificatore EXI lavora quindi con un ***insieme di grammatiche***. Durante il processo di codifica si crea quindi una pila di automi, ad ogni evento il codificatore conosce quale è l'automa attuale e quale sia lo stato dell'automa.

Quando si verifica un evento l'automa attuale cambia stato e se è necessario viene invocata la grammatica associata all'evento che si è verificato.

La codifica del singolo evento nel flusso XML è effettuata come segue:

- considera il nuovo evento da codificare;
- usa la grammatica attuale per determinare l'*event code* dell'evento;
- codifica l'*event code* seguito dall'eventuale contenuto dell'evento (codificato in base al tipo di contenuto);

- valuta la produzione della grammatica che è associata all'evento che si è verificato[5];
- ripeti finché l'evento *End Document (ED)* non viene codificato.

Con riferimento all'esempio introdotto alla sezione 4.3 la cui grammatica è visibile al listato 4.2 vediamo che la codifica avviene come segue

1. si è gia codificato l'elemento Pluto;
2. si usa la grammatica dell'elemento Pluto per codificare il prossimo evento (listato 4.2), la produzione di partenza è Attribute-pippo;
3. l'evento che si verifica è un evento *Attribute("pippo")*, ed esso viene codificato con il suo event code (l'event code è 0 poichè c'é solo una parte destra nella produzione);
4. il valore dell'attributo viene codificato (a seconda del tipo il dato 4 viene scritto);
5. la produzione viene valutata, ovvero la prossima produzione è Attribute-paperino, non ci sono grammatiche associate all'evento Attribute pippo quindi la grammatica attuale resta la stessa;
6. l'evento che si verifica è un evento *Attribute("paperino")* ed esso viene codificato con event code 0 e il dato ad esso associato viene scritto nel flusso EXI;
7. la produzione viene valutata, la prossima produzione è la produzione End-Element;
8. l'evento End-Element si verifica e viene codificato con event code 0'
9. si ritorna alla grammatica precedente.

4.6 Codifica dei tipi

Lo standard EXI stabilisce delle equivalenze tra i tipi di dato disponibili nello standard XML e i tipi di dato supportati in uno stream EXI. Viene inoltre

[5] Valutare produzione: tra le possibili parti destre della produzione attuale si seleziona quella che ha come simbolo terminale l'evento da codificare. Se all'evento è associata una grammatica quella diventa la grammatica attuale.

Tabella 4.3: Datatypes EXI (in grigio quelli implementati)

Built-in EXI Datatype Representation	XML Schema Datatypes
Binary	*base64Binary* *hexBinary*
Boolean	*boolean*
Date-Time	*dateTime* *time* *date* *gYearMonth* *gYear* *gMonthDay* *gDay* *gMonth*
Decimal	*decimal*
Float	*float,double*
Integer	*integer*
String	*string, anySimpleType, anyUri, duration, QName*
n-bit Unsigned Integer	Used by Integer datatype in some cases
Unsigned Integer	Used by Integer datatype for *unsigned integers*
List	List derived types

fornita una specifica di codifica per ognuno dei tipi di dato EXI e un possibile algoritmo per la sua implementazione.

Durante lo sviluppo della tesi si sono implementati la maggior parte dei tipi di dato previsti, con particolare attenzione a quelli presenti negli schemi XML previsti nel progetto SENSEI.

L'implementazione realizzata lascia però spazio all'introduzione di altri tipi di dato semplicemente realizzando le funzioni che ne effettuano la codifica e la decodifica.

In tabella 4.3 è possibile trovare la lista dei tipi di dato EXI con in evidenza quelli attualmente supportati.

4.6.1 Esempio di algoritmo per la decodifica di interi senza segno:

A titolo di esempio si riporta di seguito un possibile algoritmo, indicato in [9] per la decodifica degli interi senza segno.

Il tipo *Unsigned Integer* supporta interi di dimensione arbitraria, rappresentati come sequenze di *byte* terminate da un byte il cui *bit* più significativo è 0. Il valore dell'intero è salvato nei 7 *bit* meno significativi.

1. cominciare con valore iniziale = 0 e moltiplicatore iniziale = 1;
2. leggere il prossimo *byte*;
3. moltiplicare il valore dell'intero senza segno presente nei 7 *bit* meno significativi per il moltiplicatore corrente e sommare il risultato al valore attuale;
4. moltiplicare il moltiplicatore per 128;
5. se il *bit* più significativo è 1 tornare al passo 2.

Semplici algoritmi di questo tipo di usano per la codifica/decodifica di tutti i tipi EXI.

4.7 EXI grammars

Si ricordi che le grammatiche EXI sono grammatiche regolari, le cui produzioni sono associate a degli *event codes*. Un codificatore EXI, guidato dallo stream degli eventi XML, associa gli eventi a delle produzioni grammaticali ed usa l'*event code* della produzione associata per rappresentare la struttura del documento XML. Poichè le grammatiche EXI sono grammatiche regolari, la sequenza degli *event codes* scritta da un codificatore corrisponde al percorso da seguire nel *automa a stati finiti* che accetta la grammatica in esame.

È importante notare che in realtà, dato che XML non è un linguaggio regolare, una sola grammatica EXI non può essere usata per rappresentare un intero flusso XML, il codificatore lavorerà quindi con una pila di grammatiche, una per ogni *element content model*, una cioè ***per ogni elemento*** definito nello schema del documento XML in esame.

Listato 4.3: Esempio di grammatica EXI

```
Productions
```

```
LeftHandSide 1 :
        Terminal 1    NonTerminal 1
        Terminal 2    NonTerminal 2
        Terminal 3    NonTerminal 3
        Terminal 4    NonTerminal 4
        Terminal 5    NonTerminal 5
        Terminal 6    NonTerminal 6

LeftHandSide 2 :
        Terminal 1    NonTerminal 1
        Terminal 2    NonTerminal 2
        Terminal 3    NonTerminal 3
```

La specifica EXI stabilisce che la generazione di queste grammatiche deve avvenire *run-time*, ovvero che l'informazione sulla grammatica non è in effetti contenuta nel pacchetto EXI. Il codificatore EXI, nel processo di generazione delle grammatiche, può usare dell'informazione aggiuntiva circa la struttura del documento XML da codificare (come ad esempio un documento di tipo XSD), o può evincerla direttamente dal documento.

In [9] si teorizzano 2 tipi di grammatiche EXI:

> *Le* **built-in XML grammars** *sono dinamiche ed evolvono continuamente in base a quanto il codificatore apprende durante la codifica dello stream EXI. Nuove grammatiche sono create per descrivere il contenuto di nuovi elementi e nuove produzioni sono aggiunte per rifinire le grammatiche esistenti.*
>
> *Le* **Schema-informed grammars** *accettano tutti i documenti XML indipendentemente se e quanto bene rispettano lo schema che esse rappresentano. Se gli eventi che si verificano sono prevista dalla grammatica schema-informed essa viene usata per codificarli, in altro caso si utilizzano le built-in XML grammars. In generale, eventi per i quali una grammatica schema-informed è presente saranno codificati in maniera più efficiente.*

Si teorizza quindi una modalità "ibrida" per la generazione delle grammatiche, l'implementazione qui esposta fa tuttavia una scelta più restrittiva e lavora solo in modalità ***schema-informed***, suppone cioè che venga fornito un documento XSD che contiene la ***completa*** informazione sulla struttura del documento XML.

Nel capitolo 5 vengono trattate in dettaglio le motivazioni di questa scelta, da subito si può però capire che lavorando in modalità *schema-informed* le

grammatiche possono essere generate a priori, e quindi riutilizzate nel caso di documenti XML che usino lo stesso schema.

Per aiutare la comprensione si riporta uno schema XML di esempio e lo *stack* di grammatiche ad esso associato nel caso di codifica in modalità *schema-informed*.

Listato 4.4: Schema XML esempio

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="notebook">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="note" type="Note"/>
      </xs:sequence>
      <xs:attribute ref="date" />
    </xs:complexType>
  </xs:element>
  <xs:complexType name="Note">
    <xs:sequence>
      <xs:element name="subject" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
    <xs:attribute ref="date" use="required" />
    <xs:attribute name="category" type="xs:string"/>
  </xs:complexType>
  <xs:attribute name="date" type="xs:date"  />
</xs:schema>
```

In figura 4.4 vediamo come il codificatore EXI che sia in possesso dell'informazione contenuta nello schema precedente la possa usare per produrre le grammatiche da usare durante il processo di codifica. Nello specifico possiamo vedere l'automa associato al linguaggio regolare che rappresenta l'elemento "Note".

4.8 Generazione delle Grammatiche

Si descrive ora il processo di generazione delle grammatiche a partire da un documento XML schema (XSD).

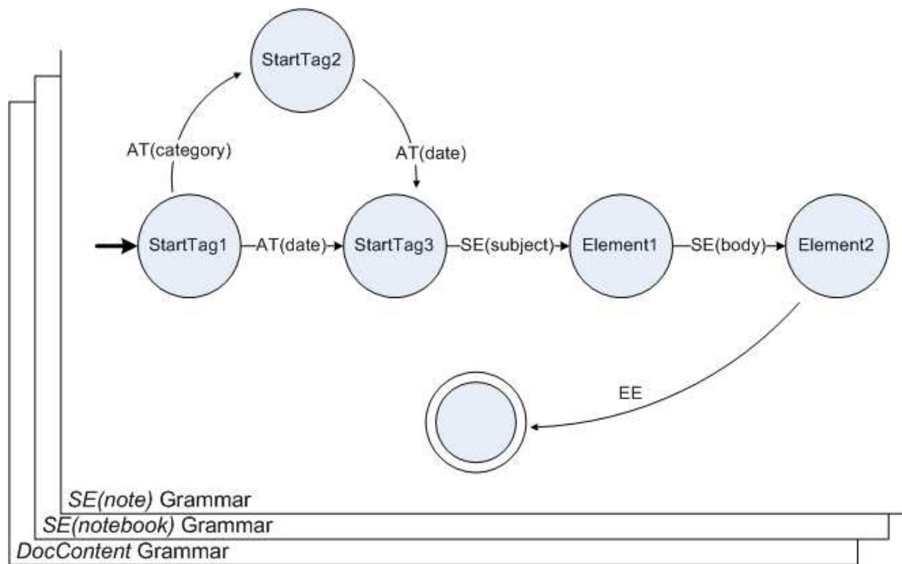

Figura 4.4: Stack di grammatiche riferito listato a 4.4 e tratta da [10]

Deve essere creata una grammatica per ogni *element declaration*[6] presente nello schema.

> *Quando uno più schemi XML sono disponibili per definire il contenuto di uno stream EXI, una schema-informed element grammar Element$_i$ è derivata per ogni element declaration E_i descritta negli schemi, dove $0 \leq i < n$ e n è il numero di element declaration presenti negli schemi.*

Ogni *schema-informed element grammar* è quindi generata con i seguenti passi:

1. creare una *proto-grammar*[7] che descrive il *content model*[8] secondo l'informazione contenuta nello schema;

2. normalizzare la *proto-grammar* per renderla una *EXI grammar*;

3. assegnare gli event code ad ogni produzione nella *EXI grammar*;

[6] Con element declaration si intende ogni elemento dello schema che sia del tipo xs:element

[7] Con proto-grammar si intende una grammatica EXI che è in fase di creazione e che non ha ancora subito il processo di normalizzazione

[8] Il termine content model indica la tipologia dell'element declaration associato alla grammatica creata, sia esso un tipo semplice (che contiene dati) o un tipo complesso (che contiene altri element)

4. aggiungere produzioni alla *EXI grammar* per rappresentare eventi EXI che possono accadere ma che non sono previsti dallo schema (non implementato).

Una *proto-grammar* contiene produzioni nella forma:

Listato 4.5: Grammatica EXI non normalizzata

```
LeftHandSide :
        Terminal 1 RightHandSide 1
        RightHandSide 2
```

sul lato right-hand side possono essere presenti altre produzioni e non necessariamente simboli terminali.

Una *EXI grammar* contiene invece ***necessariamente*** produzioni che sul lato destro hanno un simbolo terminale e ***opzionalmente*** un simbolo non terminale.

4.8.1 Passo 1

Come si vede nel listato 4.6 la grammatica di un elemento è costituita dalle grammatica di tipo ad essa associata. Innanzitutto si devono quindi creare le *type grammar*[9] associate ad ogni elemento dello schema. Lo standard definisce quindi per ogni tipo (primitivo, semplice o complesso), di elemento presente nello schema l'algoritmo che crea le *proto-grammar* ad esso associate (il loro numero varia a seconda del tipo).

Viene anche prototipato un operatore *somma di grammatiche* $\oplus$ che serve per effettuare l'unione di due grammatiche, in un processo di tipo *bottom-up* le grammatiche generate vengono quindi "fuse" tra loro fino ad ottenere l'unica grammatica che caratterizza l'*element declaration* in esame.

Listato 4.6: Grammatica di un Elemento EXI

```
Element i , 0 :
        Type j , 0
```

4.8.2 Passo 2

Le grammatiche finora create vengono chiamate *proto-grammars* perché a causa di questo processo di creazione non sono ancora ***normalizzate***, hanno cioè al loro interno produzioni che non contengono sul lato destro *simboli*

[9]Si definisce type grammar una grammatica le cui produzioni sono generate sulla base delle informazioni di tipo dell'elemento a cui essa fa riferimento

terminali, semplificando si può dire che queste grammatiche hanno produzioni che non coinvolgono eventi ma solo altre produzioni; sono inoltre presenti produzioni "inutili" perchè non raggiungibili.

La specifica prevede quindi un algoritmo di ***normalizzazione*** che porta ad ottenere grammatiche e quindi automi, ottimizzati.

> *Ogni produzione in una grammatica EXI normalizzata ha esattamente un simbolo non terminale sul lato sinistro e un simbolo terminale per ogni riga sul lato destro, seguito opzionalmente da al più un simbolo non terminale. Inoltre le grammatiche EXI normalizzate non contengono 2 produzioni con lo stesso simbolo non terminale sul lato sinistro e lo stesso simbolo terminale sul lato destro. Questa normalizzazione è una forma ristretta della forma normale di Greibach [**?**].*

La normalizzazione avviene in 2 passi:

1. eliminazione delle produzioni che non hanno simboli terminali;

2. eliminazione dei simboli terminali duplicati.

4.8.3 Passo 3

Una volta che i passi precedenti sono stati completati alla grammatica normalizzata vanno aggiunti gli *event codes*.

Per ogni produzione della grammatica si assegna un *event code* univoco. Data una grammatica EXI normalizzata G_i si applica il seguente procedimento ad ogni simbolo non terminale $G_{i,j}$ che si trova sul *left hand side*.

1. ordinare tutte le produzioni con $G_{i,j}$ sul *left hand side*, prima per tipologia di evento (AT, SE, EE, CH) , e poi internamente per ordine lessicografico.

2. assegnare gli event code in ordine crescente, a partire da 0 e fino a n, con n numero di produzioni sulla *right hand side*

Una generica grammatica normalizzata alla quale sono stati assegnati gli *event codes* è mostrata nel listato 4.7

Listato 4.7: Grammatica EXI normalizzata con event codes

```
Grammar 1                                    Event Code
LeftHandSide 1_1  :
```

```
        Terminal 1 RightHandSide 1_1     0
        Terminal 2 RightHandSide 1_2     1
LeftHandSide 1_2 :
        Terminal 1 RightHandSide 1_1     0
        Terminal 2 RightHandSide 1_2     1
        Terminal 3 RightHandSide 1_3     2
```

4.9 Il processo di Codifica

Vediamo ora un semplice esempio che mostra i passi di funzionamento del codificatore EXI implementato.

Consideriamo lo schema XSD che si trova nei listati 4.8 e 4.9, essi definiscono una classe di documenti XML di cui il documento XML al listato 4.10 è un esempio.

È uno schema presente nel progetto SENSEI che definisce la struttura dei documenti XML che vengono scambiati all'interno della rete per determinare quale sia lo stato dei LED dei sensori.

Come si vede al listato 4.10 2 sono gli elementi contenuti nel documento. `RDF` indica il *Resource Definition Format*, sta quindi ad indicare che il documento descrive una risorsa dei nodi. `LED` indica ovviamente il nome della risorsa che, come si vede anche dallo schema XSD, ha 2 possibili attributi: `about` e `hasBooleanValue` che indicano rispettivamente l'indice di lettura (numero intero positivo) e lo stato del led (valore booleano appunto).

Listato 4.8: Schema di tipo xsd Actuators.rdf

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="urn:sensei:rai"
xmlns:rdf2="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:res="urn:sensei:rai">
  <xs:import namespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  schemaLocation="ActuatorsRDF.xsd"/>

  <xs:element name="LED">
    <xs:complexType>
      <xs:attribute ref="rdf2:about" use="required"/>
      <xs:attribute name="hasBooleanValue" use="required"
      form="qualified" type="xs:boolean"/>
    </xs:complexType>
  </xs:element>


</xs:schema>
```

Listato 4.9: Schema di tipo xsd ActuatorsRDF.rdf

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
targetNamespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdf2="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:res="urn:sensei:rai">
  <xs:import namespace="urn:sensei:rai" schemaLocation="Actuators.xsd"/>
```

```
  <xs:element name="RDF">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="res:LED" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:attribute name="about" type="xs:nonNegativeInteger"/>
</xs:schema>
```

Listato 4.10: Documento XML da codificare

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf2:RDF xmlns:rdf2="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:res="urn:sensei:rai">
  <res:LED rdf2:about="2" res:hasBooleanValue="1"/>
</rdf2:RDF>
```

Le grammatiche generate per questo schema sono 3:

- Document Grammar: la grammatica del documento, che è sempre presente;
- G-rdf2:RDF: la grammatica dell'elemento RDF;
- G-res:LED: La grammatica dell'elemento LED.

La loro struttura si vede nel listato 4.11

Listato 4.11: Documento XML da codificare

```
rdf2_DocumentGrammar:
Document:
          SD DocContent
DocContent:
          SE(rdf2:RDF) DocEnd
DocEnd:
          ED

G-rdf2:RDF:
Term-res:LED-0-0:
          SE(res:LED) Term-res:LED-0-1
          EE
Term-res:LED-0-1:
          SE(res:LED) Term-res:LED-0-1
          EE

G-res:LED:
Attribute-rdf2:about-0:
          AT(rdf2:about) Attribute-rdf2:about-1
Attribute-rdf2:about-1:
          AT(res:hasBooleanValue) Attribute-res:hasBooleanValue-1
Attribute-res:hasBooleanValue-1:
          EE
```

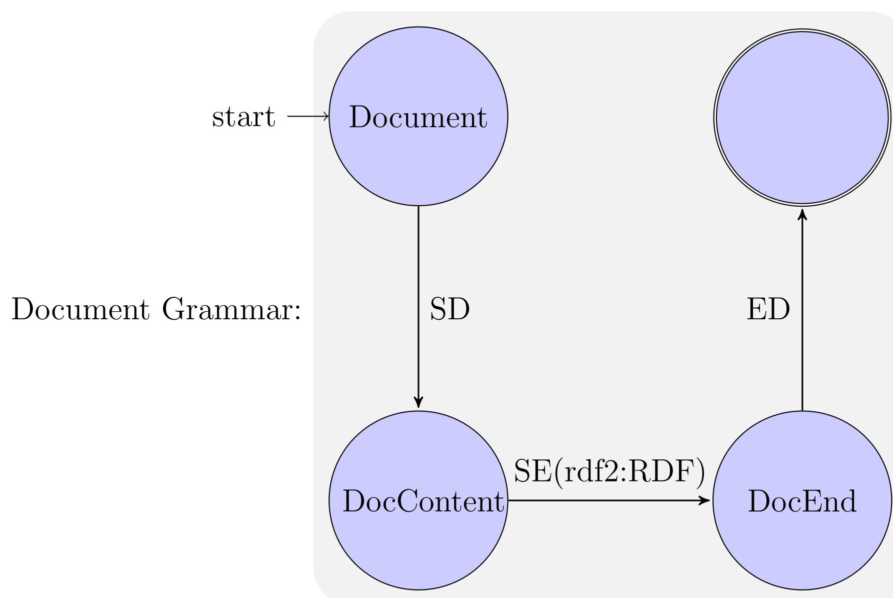

Figura 4.5: Document Grammar

Nelle figure 4.5,4.6,4.7 si vedono gli automi associati a ciascuna delle 3 grammatiche:

Infine la figura 4.8 mostra gli eventi prodotti dal documento XML al listato 4.10 e il conseguente pacchetto EXI prodotto.

Se si ripercorre la sequenza degli eventi si può vedere come la codifica lavori sui 3 automi. Ad ogni evento SE la grammatica dell'elemento associato viene invocata, si carica cioè l'automa ad essa associato. Al raggiungimento dello stato EE di un automa si torna all'automa precedente che ne aveva invocato il caricamento.

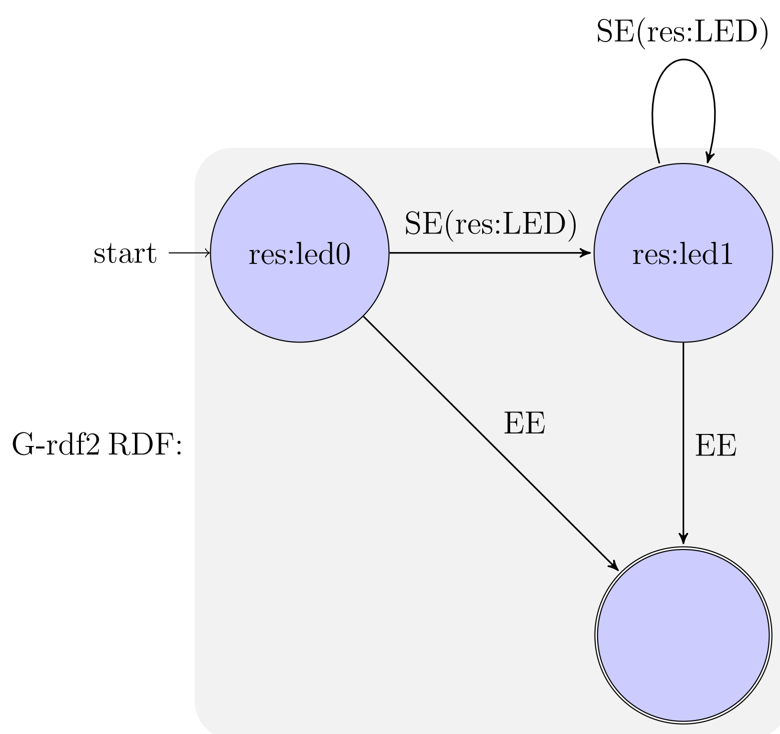

Figura 4.6: RDF element Grammar

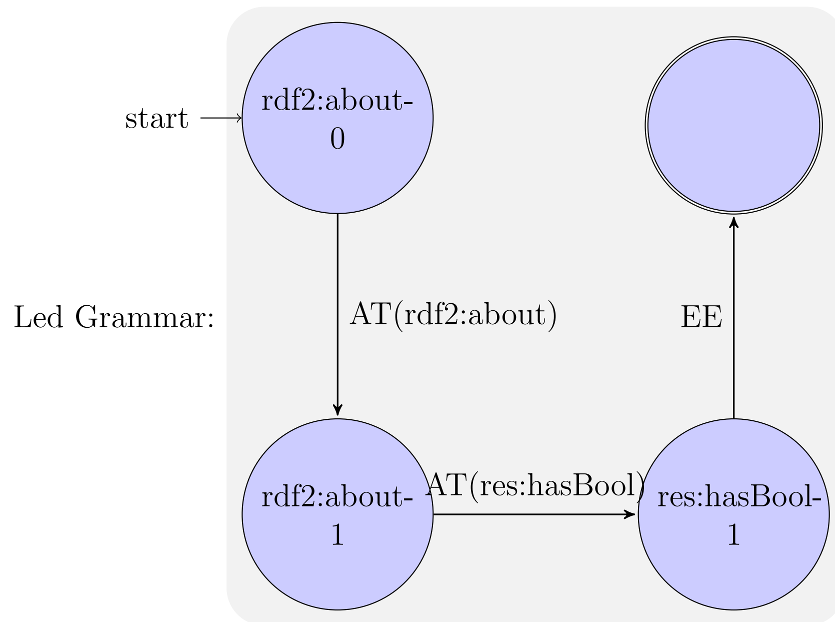

Figura 4.7: Led Grammar

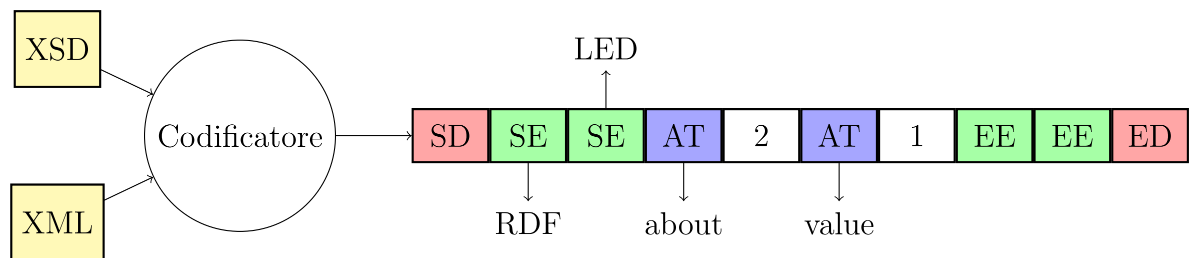

Figura 4.8: Esempio di codifica EXI

Capitolo 5

L'applicazione sviluppata

Questo capitolo descriverà il contributo di questa tesi sullo scenario XML-EXI, le motivazioni e i requisiti da cui trae origine, le scelte implementative che sono state fatte ed infine la struttura della libreria sviluppata. Si cercherà anche di dare un dettaglio delle ottimizzazioni utilizzate, soprattutto per la parte di codice scritta per i sensori, alcune riflessioni possono risultare utili anche in progetti in ambiti diversi che soffrano degli stessi stringenti requisiti *hardware*.

L'implementazione sviluppata prevede l'implementazione di un codificatore/decodificatore EXI che lavora in due fasi. In una prima fase un componente detto *preprocessore* si occupa della generazione delle grammatiche EXI a partire dall'informazione contenuta negli schemi XSD. In una seconda fase il componente detto *processore* si occupa della codifica/decodifica di tutti i documenti XML che rispettano uno degli schemi su cui è stato istruito dal preprocessore. Questo tipo di architettura richiede quindi di conoscere a priori gli schemi di riferimento per i documenti che si vorranno codificare ma consente di produrre un *processore* EXI molto compatto e per questo particolarmente adatto ad essere utilizzato nelle reti di sensori.

Nello specifico la sezione:

- Assunzioni iniziali, tratta l'ambito applicativo ed i requisiti derivanti dalla struttura *hardware* e *software* dei nodi.

- Scelte implementative, dá una visione generale della soluzione proposta ed implementata, tenendo conto delle esigenze di sviluppo.

- Preprocessore Ruby, esamina in dettaglio il funzionamento del Preprocessore, la sua struttura, gli input e gli output che produce, le ottimizzazioni introdotte.

- Processore C, analizza il codice prodotto per essere caricato sui nodi e sul *gateway* del progetto SENSEI.
- Ottimizzazioni, dá una panoramica delle ottimizzazioni che si sono usate nel codice prodotto, con particolare riferimento alla parte scritta in *C*.

5.1 Assunzioni iniziali

Vediamo con maggior dettaglio quali sono le scelte che sono state fatte durante l'implementazione dello standard EXI con riferimento all'ambito applicativo da cui trae origine questa tesi.

I requisiti di sviluppo di questa tesi sono molto legati all'hardware sui cui si lavora. Problemi normalmente trascurabili come ad esempio la dimensione del codice generato (oltre che dei dati) risultano di estrema rilevanza.

Un'altra cosa da evidenziare è che il progetto SENSEI prevedeva già prima dell'inizio della tesi l'utilizzo di XML ed EXI; sono stati quindi da subito disponibili gli schemi XSD che il codificatore avrebbe dovuto supportare per la sua integrazione nel progetto.

Queste informazioni non hanno però impedito lo sviluppo di un codificatore EXI generico, ma hanno sicuramente condizionato alcune scelte implementative come ad esempio quella sui tipi di dato supportati, o quelle relative alla struttura dei dati in memoria.

Nei listati in appendice A si possono vedere i principali schemi di riferimento che si sono usati durante lo sviluppo della libreria.

I tipi di dato supportati sono quindi:

- `boolean`
- `unsigned integer`
- `float`
- `decimal`
- `string`
- `integer`

Gli elementi XSD che il codificatore attualmente supporta sono invece:

- Simple Element (elementi semplici es `<pluto>3</pluto>`)
- Complex Element (elementi complessi es `<pluto pippo=4 />`)

- Element Content (i valori contenuti in elementi semplici)
- Attribute (gli attributi di elementi complessi)
- Sequence (sequenze di elementi)

Riassumendo, i fattori che hanno principalmente condizionato le scelte di sviluppo sono quindi stati:

- memoria rom dei nodi (solo 48KB)
- memoria ram dei nodi (solo 10KB)
- tempi di elaborazione
- nozione a priori degli schemi da implementare
- memoria occupata dagli altri strati del sistema, tinyos e codice del progetto SENSEI

5.2 Scelte implementative

I requisiti cogenti al progetto hanno fatto propendere per un implementazione parziale, ma comunque compatibile con lo standard.

In particolare al scelta più limitante riguarda il supporto alla sola modalità ***schema informed***. Il codificatore/decodificatore suppone quindi di essere sempre in possesso dell'informazione, ottenuta mediante documento XSD, sulla struttura del documento XML da elaborare.

Le grammatiche, e gli automi che le implementano, che caratterizzano un *set* di documenti XML vengono quindi generate *offline*.

Il codice prodotto è quindi diviso in due parti:

1. un ***preprocessore***, scritto in *Ruby*, che si occupa di ricevere gli schemi relativi agli insiemi di documenti XML che si vogliono supportare e produrre gli *input* per il processore, contenenti le grammatiche scritte in linguaggio *C*.
2. un ***processore***, scritto in *C*, che contiene il codice statico che implementa la pila di automi e riceve in compilazione le grammatiche generate dal preprocessore.

Un'altra scelta importante riguarda l'idea di definire una particolare formattazione della memoria che distribuisca i dati contenuti nel documento

XML da codificare in una struttura dati più compatibile alla logica di lavoro del *C*.

La codifica implementata, inoltre, lavora solo nella modalità ***byte aligned***, nella quale ogni elemento dello stream EXI occupa il numero di *byte* più basso possibile, la modalità ***bit aligned*** non è stata invece implementata perchè, anche se avrebbe ridotto ulteriormente la dimensione dello *stream* EXI si è valutato che il *trade-off* con i costi di implementazione fosse svantaggioso ai fini del progetto.

Lo schema di lavoro del codice risultante è visibile in figura 5.1.

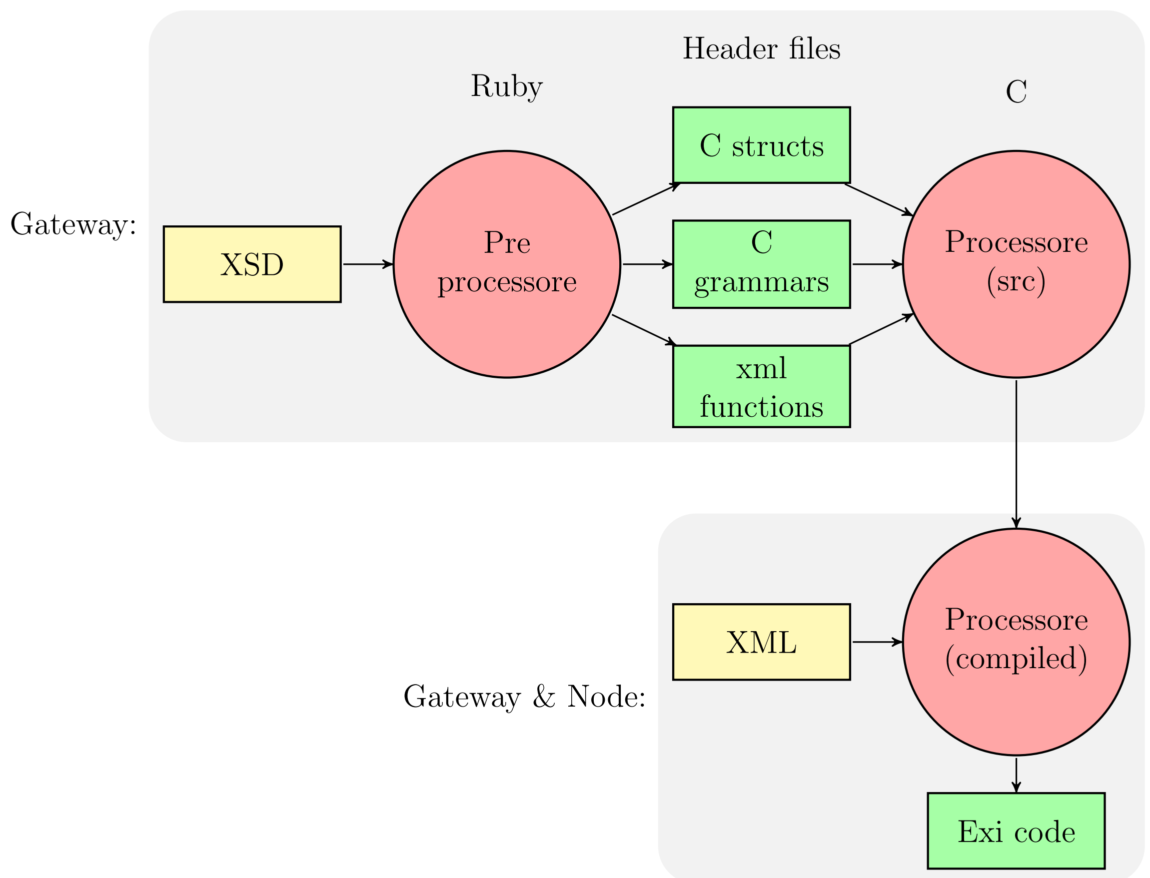

Figura 5.1: Schema di funzionamento

Si può quindi vedere come il codice del preprocessore sia stato pensato per essere eseguito su *hardware* con architettura *x86*, mentre il codice del processore, ovvero il *core* della libreria, è tale da poter essere compilato su ar-

chitetture a 32bit e 64bit, come i normali *computer* ma anche su architetture a 16bit quali i sensori impiegati nel progetto SENSEI.

L'utilizzo della libreria avviene quindi in due fasi: Inizialmente il preprocessore riceve gli schemi delle classi di documenti XML che si vogliono codificare/decodificare in una certa applicazione e produce le grammatiche EXI ad essi corrispondenti. I dati elaborati sono quindi inclusi nel processore.

Il punto di interfacciamento tra *preprocessore* e *processore* sono, come si vede dalla figura, dei file *header*, file sorgente *C* che contengono principalmente le definizioni delle strutture dati e degli array che rappresentano le grammatiche prodotte e, più in generale, l'informazione che il preprocessore ha estrapolato dallo schema XSD.

Il codice sorgente del processore, completato con questa informazione, può essere ora *compilato*, congiuntamente con il codice che utilizza la libEXI e caricato sui nodi o eseguito sul *gateway*.

L'evidente vantaggio di questa struttura è che una volta stabiliti gli schemi da supportare non è più necessario invocare il preprocessore, che andrà però utilizzato nuovamente nel momento in cui essi dovessero cambiare.

5.2.1 Il formato della memoria

Un'altra importante decisione presa durante la pianificazione della libreria è stata quella di creare un formato alternativo per la memorizzazione dei dati contenuti nei documenti XML. Un codificatore EXI deve infatti produrre a partire da documenti XML documenti EXI e viceversa. Nello scenario applicativo in esame non è però necessario, né realizzabile a causa delle limitazione *hardware*, che nei sensori wireless il processo di codifica/decodifica del formato XML sia completo (che si decodifichi cioè per esteso il documento XML all'interno del nodo).

Si è quindi prototipato ed implementato un formato intermedio (chiamato da ora *memoria C*) per la memorizzazione dei dati contenuti nei documenti XML. Il documento XML viene letto tramite un processore XML e i dati in esso contenuti sono scritti in un area di memoria contigua, seppur mantenendo l'informazione, anche di struttura, in essi contenuta.

Le porzioni di quest'area di memoria prendono significato perchè possono essere interpretate come insiemi di *byte* che appartengono a strutture dati note. Queste strutture dati sono oggetti di tipo *struct* del linguaggio *C*. Possono contenere uno o più dei tipi primitivi supportati da EXI o dei puntatori ad altre zone della memoria, corrispondenti ad altre strutture dati, mantenendo la struttura “ad albero” tipica dei documenti XML.

Le definizioni di queste *struct* sono generate dal preprocessore durante la fase di elaborazione dello schema, contemporaneamente quindi alla gene-

razione delle grammatiche. Il codice C conosce queste informazioni ed è in grado di dare significato alla memoria che viene prodotta.

L'utilizzo di questo formato intermedio è quello che si può vedere in figura 5.2. Il documento XML viene tradotto in *memoria C* nel *gateway* e in seguito nel formato EXI. Nodo e *gateway* comunicano solo mediante il formato EXI. Il nodo a sua volta può accedere ai dati decodificando il pacchetto EXI.

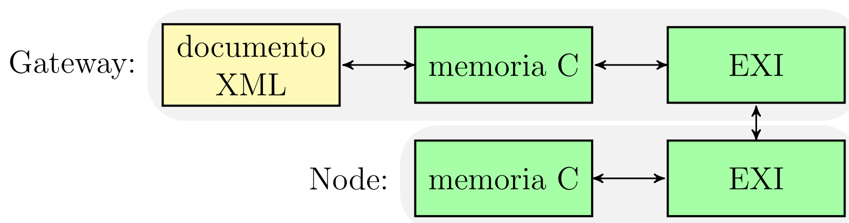

Figura 5.2: Schema comunicazione della memoria

Il codificatore EXI riceve quindi i dati in ingresso tramite un puntatore ad un area di memoria opportunamente preparata con i dati scritti in questo formato e viceversa il decodificatore EXI produce output in questo formato intermedio.

Il diagramma 5.2 evidenzia come sul nodo l'operazione di codifica/decodifica EXI sia limitata all'uso della memoria C, nessun documento XML viene effettivamente processato sul nodo.

Si vede ora con un esempio come è strutturata la suddetta memoria.

Lo schema 5.1 produce le struct visibili in 5.2.

Listato 5.1: Schema di tipo XSD ActuatorsRDF.rdf

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   ...>
  <xs:element name="RDF">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="res:LED" minOccurs="0" maxOccurs="
            unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:attribute name="about" type="xs:nonNegativeInteger"/>

  <xs:element name="LED">
    <xs:complexType>
      <xs:attribute ref="rdf2:about" use="required"/>
      <xs:attribute name="hasBooleanValue" use="required" form="
          qualified"
```

```
        type="xs:boolean"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listato 5.2: Struct corrispondenti allo schema precedente

```
struct res_LED{
    xs_uint_32 rdf2_about;
    xs_bool res_hasBooleanValue;
} __attribute__((packed));
struct rdf2_RDF{
    void * res_LED;
} __attribute__((packed));
```

Come già analizzato al capitolo 4 lo schema definisce infatti un elemento di tipo `RDF` che contiene 0 o più elementi di tipo *LED*. L'elemento `LED` è formato da due attributi con tipi di base EXI.

Si consideri il documento di esempio al listato 5.3.

Listato 5.3: Documento XML da codificare

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf2:RDF>
  <res:LED rdf2:about="1" res:hasBooleanValue="1"/>
  <res:LED rdf2:about="2" res:hasBooleanValue="0"/>
</rdf2:RDF>
```

La figura 5.3 contiene un esempio di come il documento 5.3 venga *memorizzato* in *memoria C*.

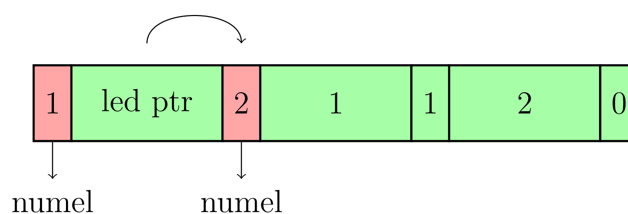

Figura 5.3: Rappresentazione in memoria del documento XML

Si vede come la memoria contenga una istanza della struct `struct RDF` e due istanze della `struct LED`. In rosso sono evidenziati i *byte* etichettati come `numel` che hanno lo scopo di contare quante istanze sono presenti per ogni *struct*. Questi valori sono infatti necessari in fase di inizializzazione della memoria per assegnare correttamente i puntatori della *struct* principale, ed

in fase di lettura dei dati, poichè solo sapendo quanti elementi sono presenti per ogni *struct* diversa è possibile leggerli correttamente.

Il puntatore contenuto nella `struct RDF` punta quindi al valore `numel` che indica l'inizio delle istanza della `struct LED`.

Si noti che questo modo di strutturare la memoria consente vari livelli di annidamento, sono quindi supportati dal codificatore EXI schemi che contengono più elementi annidati come quello al listato 5.4

Listato 5.4: Documento XML da codificare

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf2:RDF>
  <res:LEDgroup>
        <res:LED rdf2:about="1" res:hasBooleanValue="1"/>
        <res:LED rdf2:about="2" res:hasBooleanValue="0"/>
  </res:LEDgroup>
</rdf2:RDF>
```

Il codice prodotto fornisce delle funzioni *helper* utili per lavorare correttamente con la memoria, e dialogare quindi con il codificatore EXI che vengono introdotte in 5.4.2.

5.2.2 Il formato delle grammatiche

L'altro importante output prodotto dal preprocessore *Ruby* sono appunto le grammatiche relative agli automi con i quali verrà processato il documento XML. Il funzionamento del codice *C* che processa gli automi è spiegato in 5.4. Si da però qui una descrizione del formato utilizzato negli *header file* prodotti dal preprocessore.

Il singolo automa viene rappresentato come un *Array* tridimensionale le cui dimensioni rappresentano rispettivamente *left-hand side, right-hand side, (event,next-prod,skip-size,next-grammar)*, le dimensioni di questo *array* sono dunque $n_prod * n_rside * 4$ dove *n_prod* indica il numero di produzioni della grammatica e *n_rside* il numero massimo di produzioni sul lato destro all'interno delle produzioni precedenti.

Nella dimensione "più interna" sono invece sempre presenti 4 valori:

- event type: il tipo di evento a cui corrisponde la produzione;
- next-prod: la prossima produzione, ovvero lo stato successivo dell'automa;
- skip-size: valore che indica il numero di elemento all'interno della struct attuale;

- next-grammar: l'automa che deve essere invocato durante il processamento di questa produzione, o alternativamente la funzione di codifica/decodifca del dato.

Nuovamente è utile vedere un esempio. Con riferimento al listato 5.1 vediamo le grammatiche prodotte in 5.5 e il loro equivalente C in 5.6.

Listato 5.5: Grammatiche relative allo schema LED

```
rdf2_DocumentGrammar:
Document:
        SD DocContent
DocContent:
        SE(rdf2:RDF) DocEnd
DocEnd:
        ED

G-rdf2:RDF:
Term-res:LED-0-0:
        SE(res:LED) Term-res:LED-0-1
        EE
Term-res:LED-0-1:
        SE(res:LED) Term-res:LED-0-1
        EE

G-res:LED:
Attribute-rdf2:about-0:
        AT(rdf2:about) Attribute-rdf2:about-1
Attribute-rdf2:about-1:
        AT(res:hasBooleanValue) Attribute-res:hasBooleanValue-1
Attribute-res:hasBooleanValue-1:
        EE
```

Listato 5.6: Grammatiche C relative allo schema LED

```
xchar rdf2_DocumentGrammar[3][1][4] =
{
        {
                {START_DOCUMENT,1,0,13}
        },
        {
                {START_ELEMENT,2,0,11}
        },
        {
                {END_DOCUMENT,0,0,13}
        }
};

xchar grammar_rdf2_RDF[2][2][4] =
```

```
{
        {
                {START_ELEMENT,1,0,12},
                {END_ELEMENT,0,0,13}
        },
        {
                {START_ELEMENT,1,0,12},
                {END_ELEMENT,0,0,13}
        }
};

xchar grammar_res_LED[3][1][4] =
{
        {
                {ATTRIBUTE,1,0,TYPE_UINT}
        },
        {
                {ATTRIBUTE,2,1,TYPE_BOOLEAN}
        },
        {
                {END_ELEMENT,0,0,13}
        }
};
```

Dai due listati precedenti si può vedere come ci sia una corrispondenza diretta tra le grammatiche definite nella specifica vista al capitolo 4 e la loro implementazione nel codice *C*. Ad ogni grammatica EXI corrisponde un *array* tridimensionale, al primo livello di annidamento troviamo le produzioni complete (o *left hand side*), mentre livello più interno troviamo i 4 campi precedentemente introdotti, in particolare il primo di essi corrisponde al *simbolo terminale* che si può vedere nella corrispondente grammatica EXI.

Il preprocessore produce inoltre altri *array* e *struct* ausiliarie necessarie al funzionamento del processore, dato che il codificatore supporta la presenza contemporanea di più schemi XML nella memoria.

5.3 Il preprocessore Ruby

Il *preprocessore* è un *software* che è pensato per essere eseguito su macchine con architettura *x86* come i *gateway* del progetto SENSEI e non su i nodi. Si occupa della elaborazione dell'informazione degli schemi XSD che descrivono la struttura dei documenti XML che verranno elaborati dal processore EXI.

Ricordiamo infatti che, dati i requisiti di progetto, si è scelto di implementare un codificatore EXI che lavori nella sola modalità ***schema informed***. Le grammatiche e tutta l'informazione necessaria alla codifica di uno sche-

ma, viene quindi prodotta *off-line* e poi passata al codificatore EXI in fase di compilazione, come già visto in figura 5.1.

Scopo del preprocessore è quindi, a partire da un documento di tipo XML schema, produrre una serie di *header file* scritti in linguaggio *C* che possano essere compilati insieme al *core* statico del processore EXI vero e proprio aggiungendo l'informazione necessaria ad effettuare il processamento di documenti XML scritti secondo lo schema in esame.

In realtà, come vedremo, il preprocessore supporta la possilbilità di ricevere più schemi XSD, e produce quindi il codice *C* che consente il supporto a schemi multipli.

Possiamo definire il *preprocessore* un ***generatore di codice***.

La scelta di un linguaggio ad alto livello come *Ruby* per lo sviluppo di questa parte del progetto è principalmente motivata dalla complessità della realtà da modellare. Infatti, come si vedrà nel resto del capitolo, il livello di astrazione fornito da un linguaggio fortemente orientato agli oggetti ha consentito di creare delle classi che rappresentano fedelmente oggetti come un documento XSD, una Proto Grammatica, delle Struct o una Grammatica convertita in codice *C*. Già in fase di sviluppo si è inoltre verificato come l'utilizzo di un linguaggio di questo tipo produca codice più facilmente estensibile, nell'ipotesi di produrre, anche in vista di sviluppi futuri, il supporto ad altri tipi di elementi XML, o a schemi di maggior complessità, ad ora tralasciati durante lo sviluppo di questa tesi.

5.3.1 Schema di lavoro

Lo schema di funzionamento del preprocessore è visibile in figura 5.4.

La classe *Generator* riceve in ingresso uno o più schemi XSD e per ogni schema crea una instanza della classe *Schema Generator*, quest'oggetto è l'elemento centrale del processamento di uno schema. Esso si occupa di caricare l'informazione contenuta nello schema mediante la classe *Document*, che restituisce una classica struttura ad albero nella quale ogni documento è composto da elementi che hanno un padre e n figli.

Ottenuta questa informazione è possibile ricavare le *struct* che andranno a mimare la struttura del documento XML, come spiegato in 5.3.3.

L'oggetto *schema* crea inoltre una istanza della classe *Proto Grammar Factory*, che a partire dall'informazione contenuta nel documento XSD crea oggetti di tipo *Proto-Grammar*, crea cioè (come visto nel capitolo 4) una grammatica EXI per ogni elemento presente nello schema in esame, si veda la sezione 5.3.2 per un dettaglio del suo funzionamento.

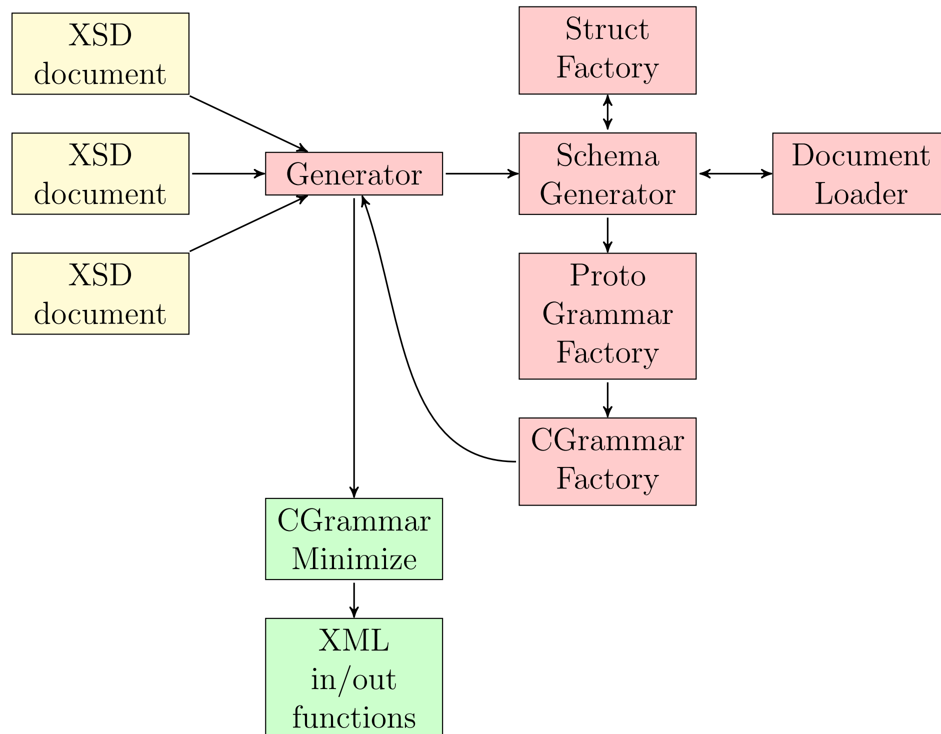

Figura 5.4: Schema di funzionamento del Preprocessore

Ottenute le grammatiche come oggetti *Ruby* è necessario effettuarne la conversione in *header file* interpretati dal compilatore *C*. Di questo si occupa la classe *CGrammar Factory* analizzata nella sezione 5.3.4.

Una volta che per ogni schema sono state ottenute le grammatiche che lo caratterizzano viene effettuata nella classe *Generator* una ulteriore ottimizzazione analizzata in 5.5. Si esaminano infatti tutte le grammatiche generate e si eliminano quelle che, pur avendo nomi diversi, contengono le stesse produzioni, caso molto frequente, ad esempio in schemi che contengono molti elementi che pur avendo nomi diversi hanno gli stessi attributi (si veda a questo proposito lo schema *Sensors* in A.2). Questa operazione di eliminazione della ridondanza consente buoni risparmi dal punto di vista della memoria, requisito primario di questo progetto.

5.3.2 Generazione Grammatiche

La classe *Proto Grammar Factory* e quelle ad essa correlate implementano le definizione contenute al capitolo 8 della specifica EXI [9]. Si occupano cioè di

creare una *Proto-Grammar EXI* per ogni elemento contenuto nello schema in esame. I passi implementati sono quelli visti in 4.8, per ogni schema si crea quindi la *Document Grammar* ad esso associata e in seguito si analizzano gli elementi dello schema.

Per ogni elemento si crea quindi una oggetto *Proto Grammar* inizialmente vuoto e poi in base al suo tipo XML si creano altri oggetti di tipo *Proto Grammar* che costituiscono le *Type Grammar* adeguate. Vengono quindi inizialmente create molte grammatiche per ogni elemento, in seguito, rispettando le regole della specifica EXI, mediante il gia visto operatore $\oplus$, implementato nella classe *Proto Grammar*, esse vengono fuse fino ad ottenere una sola grammatica per ogni elemento dello schema. Durante questa operazione le produzioni contenute nelle grammatiche di tipo vengono spostate al livello superiore ed aggiunte alla *Proto grammar* a cui sono associate.

Nella fase successiva viene effettuata la normalizzazione di ognuna delle grammatiche così create, sempre seguendo la specifica EXI si sono implementati i 2 passi necessari alla normalizzazione:

1. Eliminazione delle produzioni che non hanno simboli terminali

2. Eliminazione dei simboli terminali duplicati

In figura 5.5 vediamo la struttura dell'elemento RDF presente nello schema XSD del listato 5.1 e la *Proto Grammar* che viene creata per ogni suo componente. Solo le grammatiche relative agli elementi (in verde nella figura) verranno mantenute alla fine della normalizzazione.

In figura 5.6 è invece possibile vedere quale sia la struttura di un oggetto di tipo *Proto Grammar*.

5.3.3 Generazione Struct

La classe *Struct Factory* e quelle ad essa correlate creano, a partire dall'informazione contenuta nello schema XSD le struct che, come visto in 5.2.1, servono a dare significato all'area di memoria che contiene i dati, e la struttura, del documento XML in esame durante il processo di codifica/decodifica.

L'istanza della classe *Struct Factory* si occupa di analizzare l'oggetto di tipo Documento e di creare, per ogni elemento in esso presente, una istanza della classe *Struct*.

In base al tipo di oggetto la *struct* creata varia i suoi attributi. In particolare elementi di tipo *Complex Type*[1] vengono tradotti in *struct* contenenti

[1]Elementi il cui tipo è un tipo complesso, ovvero elementi che contengono altri elementi o attributi

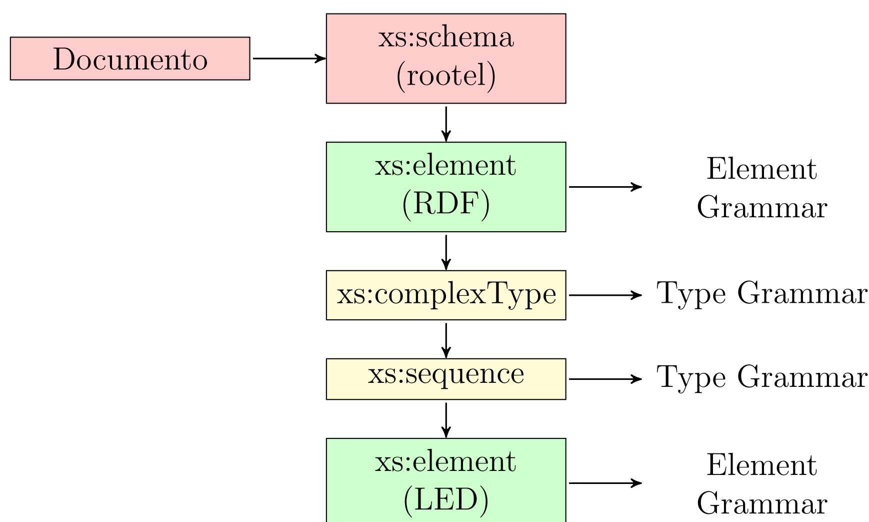

Figura 5.5: L'oggetto schema ActuatorsRDF

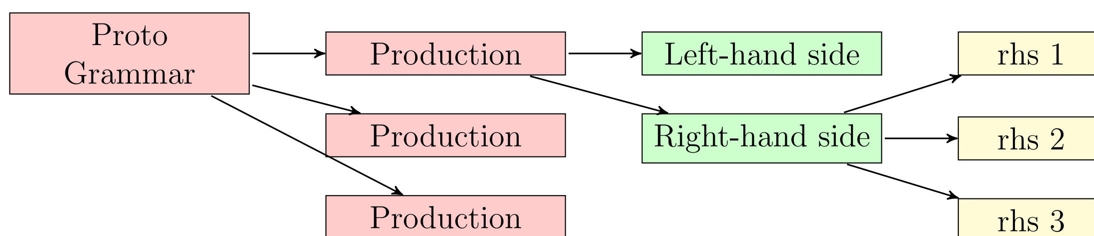

Figura 5.6: Struttura di un oggetto Proto Grammar

puntatori di tipo *void **, mentre elementi *Simple Type*[2] o *Attribute*[3] sono tradotti in *struct* che contengono i tipi di dato EXI opportuni.

5.3.4 Generazione C grammars

Nella classe *CGrammar Factory* si prendono in esame le *Proto grammar* create nelle fasi precedenti e si creano gli oggetti di tipo *CGrammar* ad esse corrispondenti.

[2] Elementi di tipo semplice, il cui tipo di dato è uno dei tipi di dato previsti dallo standard XML, questi elementi contengono quindi dati e non altri elementi

[3] Definizioni di attributi. Gli attributi sono associati a elementi, e per ogni attributo si definisce il nome e il tipo di dato all'interno di quelli previsti dallo standard XML

É quindi questa classe che produce gli oggetti che costituiscono il contenuto degli *header C* che contengono le grammatiche scritte nel formato specificato nella sezione 5.2.2.

La traduzione implica la creazione di oggetti di tipo *CGrammar* contenenti a loro volta produzioni ti tipo *CProduction* con annidate le parti destre di ogni produzione, istanze della classe *CProductionRhs*. I tipi EXI vengono mappati ne corrispondenti tipi *C* e il lato destro delle produzioni è ordinato per *event code* crescente, evitando quindi di memorizzare informazione ridondante.

Vengono inoltre prodotte altre ottimizzazioni, in particolare si minimizza il numero di grammatiche ***localmente*** allo schema in esame, fondendo in una unica istanza le *CGrammar* che hanno produzioni ***identiche***. In seguito il numero di grammatiche viene anche minimizzato ***globalmente*** fondendo le grammatiche che hanno le stesse produzioni pur appartenendo a schemi diversi.

Poichè le grammatiche sono memorizzate come semplici *array* di interi è stato necessario introdurre altre strutture dati che completassero l'informazione in essi contenuta, sia per consentire di dare significato ai dati delle grammatiche, sia per esprimere le relazioni tra una grammatica e le altre dello stesso schema

In questa fase, vengono quindi prodotto degli *array* e delle *struct* ausiliarie, le più significative sono:

- Schema struct: Contiene il puntatore al vettore delle grammatiche dello schema ed altre informazioni sulla dimensione dello schema
- All Grammar vector: È un vettore che contiene i puntatori a tutte le grammatiche presenti in memoria, indipendentemente dagli schemi di origine
- Grammar struct: Sono più strutture, una per ogni grammatica, contengono le dimensioni della grammatica e il puntatore all'array coi dati della grammatica
- Grammar vector: Ne esiste una istanza per ogni schema, contiene i riferimenti alle grammatiche dello schema.

I nomi di queste *struct* son tutti del tipo `namespace_schema`, l'utilizzatore della libreria deve, come si vedrà nella sezione 5.4, conoscere di volta in volta lo schema associato al documento XML che vuole codificare, e passarlo alle funzioni di codifica/decodifica mediante puntatore alla *schema struct* corrispondente.

5.3.5 XML helper

L'ultima tipologia di *header file* prodotto dal preprocessore contiene una serie di funzioni generate per facilitare la traduzione dei documenti XML al formato di memoria visto in 5.2.1.

In particolare, in base all'informazione contenuta negli schemi, vengono generati due *file*.

1. `xml_to_struct.h`, contiene le funzioni per leggere il contenuto di un file xml di cui lo schema è noto e trascriverlo nel formato di memoria accettato come input dal codificatore EXI;

2. `struct_to_xml.h`, contiene le funzioni per produrre un file xml a partire dall'output del decodificatore EXI.

Le funzioni generate sono ***specifiche*** per ogni schema, sono infatti generate con un nome del tipo: `void NAMESPACE_xml_to_struct_filename(char * xml_filename, void* mem_buff, int mem_size);`. Per produrre un area di memoria popolata con i contenuti di un dato documento XML è quindi sufficiente invocare la funzione `xml_to_struct_filename` per lo schema in uso nel documento.

5.4 Il processore C

Si analizza ora il funzionamento del processore EXI vero e proprio. Questa parte della tesi, come già detto, è sviluppata in *C*, con particolare attenzione all'interoperabilitá su architetture diverse. Il codice è infatti lo stesso sia per la parte che viene compilata sul gateway, sia esso una macchina ad architettura a 32 o 64 bit, sia che venga incluso all'interno di un applicazione tinyos compilata per l'architettura a 16 bit dei sensori *t-mote*.

Punto focale dello sviluppo sono stati i requisiti di sviluppo, in particolare la necessita di produrre una implementazione *estremamente* compatta in termini di dimensione: dei dati, del segmento codice e di utilizzo dello stack.

Il risultato sono un'insieme di funzioni "speculari" che si occupano, separatamente, della codifica e della decodifica dei dati.

Al centro di questa implementazione sta la funzione che fa da interprete di automi, la funzione cioè che è in grado di interpretare l'informazione contenuta nelle grammatiche prodotte dal preprocessore ed usarla per codificare/decodificare il documento XML in esame.

In figura 5.7 si vede lo schema di funzionamento del processore EXI.

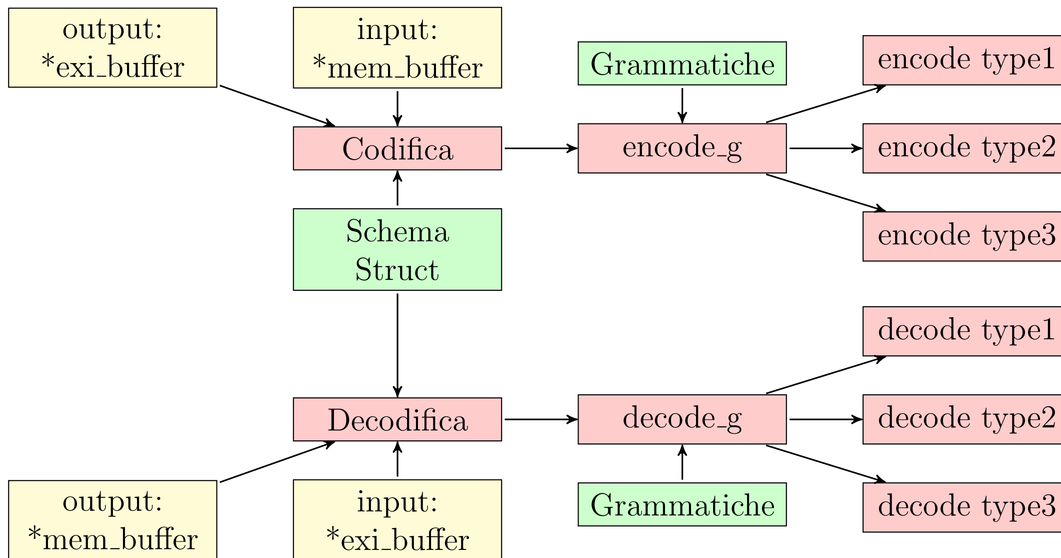

Figura 5.7: Struttura del processore EXI

La figura mostra (in rosso) le funzioni coinvolte nel processo di codifica e in quello di decodifica. Le funzioni *codifica* e *decodifica* sono quindi l'interfaccia della libreria con l'utilizzatore, come si vede i parametri di queste funzioni (in giallo) sono 2 buffer, la cui allocazione è lasciata all'utilizzatore, ed un puntatore allo schema da usare per la codifica.

Le 2 funzioni di interfaccia passano i buffer ricevuti alle funzioni che fungono da interpreti di automi, in particolare invocano la funzione *encode_g* sulla prima grammatica dello schema, la *Document Grammar*. La grammatica che contiene cioè le produzioni che definiscono inizio e fine del documento, nonchè la produzione associata allélemento piú esterno dello schema.

É importante notare come in realtà il *buffer* contenente il flusso EXI ed il puntatore allo schema attuale non siano passati a parametro alle funzioni invocate, ma assegnati a 2 puntatori globali. Le motivazioni di questa scelta sono date in 5.5, si ricordi però, durante la lettura di questo capitolo, che ogni funzione della libEXI ha accesso a questi elementi.

A questo punto lo stack di automi è stato inizializzato ed il processo di codifica può cominciare. Il nucleo della libEXI può essere infatti riassunto nella funzione *encode_g* spiegata in dettaglio in 5.4.1

Le altre funzioni che si vedono nella figura sono quelle che si occupano di realizzare la codifica di un particolare tipo di dato nel suo corrispettivo EXI o viceversa.

5.4.1 Lo stack di automi

Si analizza ora nel dettaglio in funzionamento della funzione *encode_g*, un discorso del tutto analogo si può fare per la *decode_g*.

L'idea alla base di questa implementazione è, come già evidenziato più volte, costruire un codice estremamente compatto che costituisca un interprete delle varie grammatiche che caratterizzano uno schema XSD.

Poichè, come chiarito al capitolo 4, un documento XML è caratterizzato da un insieme di grammatiche EXI è necessario disporre di uno *stack* di automi. Si è quindi scelto di creare la funzione *encode_g* che funge da interprete di un singolo automa, e di usare lo *stack* del sistema per ricordare lo stato di ogni automa durante la codifica.

Più semplicemente: nel passaggio da un automa all'altro la funzione invoca se stessa con parametri diversi.

Il difetto, proprio di tutte le implementazioni ricorsive, è quindi quello di rischiare di esaurire lo *stack*, che nei nodi è di dimensioni contenute. Tuttavia è importante sottolineare come, in tutti gli schemi del progetto SENSEI, le funzioni che la libEXI mette nello stack non siano mai più di 5, evitando quindi praticamente qualunque problema.

Guardiamo ora il prototipo della funzione:

```
void *encode_g(void * rdf_pointer, struct g_data * g_d);
```

I parametri sono, tenendo conto anche di quelli globali, i seguenti:

- buffer memoria contenente dati xml (rdf_pointer)
- buffer EXI di uscita
- puntatore alla struct *grammar struct* relativa alla grammatica in esame (g_d)
- puntatore alla schema struct

Ogni volta che viene invocata la funzione conosce quindi quale è lo schema in uso e quale è la grammatica che deve usare per processare i dati nel buffer di ingresso.

Poichè in ogni grammatica la prima produzione è in testa, la funzione comincia semplicemente dalla prima produzione, ovvero dallo stato di *start* dell'automa, a questo punto comincia un ciclo, ad ogni iterazione esamina una parte *right hand side* della produzione attuale.

Viene letto il tipo di evento relativo alla produzione in esame, mediante una struttura di tipo `switch-case` (un caso per ogni evento tra quelli gestiti in 4.2).

La produzione viene quindi esaminata in base ai dati presenti nella memoria da codificare, in particolare si usa il campo *numel* visto in figura 5.3. Se il valore è maggiore di 0 l'elemento associato alla produzione è presente. La produzione viene ***valutata***: si seleziona la prossima produzione per la grammatica corrente, e si legge il campo che contiene l'indice della prossima grammatica. Questo campo serve per accedere alla struct della grammatica successiva utilizzando il vettore delle grammatiche dello schema visto in 5.2.2.

La funzione *encode_g* viene quindi invocata un'altra volta passandole il puntatore alla nuova grammatica così selezionata e il puntatore all'inizio del segmento dati di cui si dovrà occupare la grammatica attualmente in esame.

Se invece l'evento valutato prevede che il contenuto della memoria in quel momento esaminata sia un dato, il valore del campo che solitamente contiene l'indice della grammatica successiva viene valutato come indice dei un array che contiene i puntatori alle funzioni di codifica dei tipi di dato. Il puntatore alla funzione viene quindi letto da questo array e l'opportuna funzione di codifica è invocata.

Per rendere più chiari i concetti finora esposti proseguiamo con l'esempio cominciato all'inizio di questo capitolo. Di seguito vediamo infatti il processo di codifica del listato 5.3, di cui già sono state viste le grammatiche e la struttura in memoria dei dati.

XML	grammatica	act prod	evento	next prod	next grammar
`<?xml version=1.0 encoding=UTF-8?>`	Document Grammar	0	SD	1	self
`<rdf2:RDF>`	Document Grammar	1	SE	2	rdf2_RDF
`<res:LED`	rdf2_RDF	0	SE	1	rdf2_LED
`rdf2:about=1`	rdf2_LED	0	AT	1	encode_int
`res:hasBooleanValue=1`	rdf2_LED	1	AT	2	encode_bool
`/>`	rdf2_LED	2	EE	0	null
`<res:LED`	rdf2_RDF	1	SE	1	rdf2_LED
`rdf2:about=2`	rdf2_LED	0	AT	1	encode_int
`res:hasBooleanValue=0`	rdf2_LED	1	AT	2	encode_bool
`/>`	rdf2_LED	2	EE	0	null
`</rdf2:RDF>`	rdf2_RDF	1	EE	0	null
`End of File`	Document Grammar	2	ED	0	null

Si noti che il puntatore alla *next grammar* è *null* quando gli eventi sono di tipo ED o EE. In questo caso infatti la funzione *encode_g* ritorna e, come si vede anche dalla tabella, la grammatica torna ad essere quella precedente.

Si vede anche che la codifica di questo semplice documento, simile agli altri del progetto SENSEI, coinvolge solo 3 grammatiche, il peso nello *stack* è quindi contenuto.

In figura 5.8 si può vedere una rappresentazione grafica dello stesso esempio che mostra gli stati delle invocazioni della funzione *encode_g* durante il processo di codifica. Per ogni passaggio è evidenziato il `case` in cui si trova la funzione al ricevere un certo input.

In realtà c'é una ulteriore complessità che non viene prevista dalla specifica EXI, ma si è dovuta introdurre nello sviluppo della tesi per compensare la minor informazione veicolata dal formato di memoria scelto per rappresentare i documenti XML.

Nello specifico, con riferimento alla figura 5.3, ricordiamo che ci sono principalmente 2 tipologie di *struct*: quelle che contengono puntatori ad altri punti dell'area di memoria, e che mimano quindi elementi XML che contengono altri elementi; quelle che contengono invece tipi di dato e che si riferiscono quindi a elementi XML "foglie".

Per Entrambe queste tipologie di elementi l'evento EXI associato è l'evento *start element*, tuttavia nella nostra implementazione, come si vede nel listato 5.6, i tipi di evento associati all'inizio di un elemento sono 2, questo perchè cosi facendo nel `case` che gestisce questi eventi è possibile sapere se i dati contenuti nella porzione della memoria sotto esame vanno interpretati come puntatori ad altre aree della memoria o come dati da codificare.

Della distinzione tra questi due tipi di eventi si occupa il processore, che essendo in possesso di tutta l'informazione dello schema è in grado, nel processo di traduzione delle grammatiche da grammatiche EXI a grammatiche scritte in formato C, di introdurre questa ulteriore distinzione in modo trasparente.

5.4.2 Le funzioni di utilizzo della memoria (API)

Alla luce delle precedenti considerazioni si può capire come la preparazione della memoria da passare al codificatore sia un passaggio delicato.

Per facilitare questo processo, oltre alle funzioni viste in 5.3.5 che vengono generate dinamicamente dal preprocessore, la libreria contiene altre 3 funzioni, che sono invece statiche. Utili per la creazione e manipolazione del formato di memoria usato dal codificatore/decodificatore EXI.

Queste Application Programming Interface (API) sono usate sia da quelle generate automaticamente dal preprocessore, sia dal codice che attualmente usa la libEXI all'interno del progetto SENSEI.

Il loro utilizzo facilita la preparazione della memoria e quindi l'uso del *core* della libreria, ma ha dei costi non trascurabili dal punto di vista del-

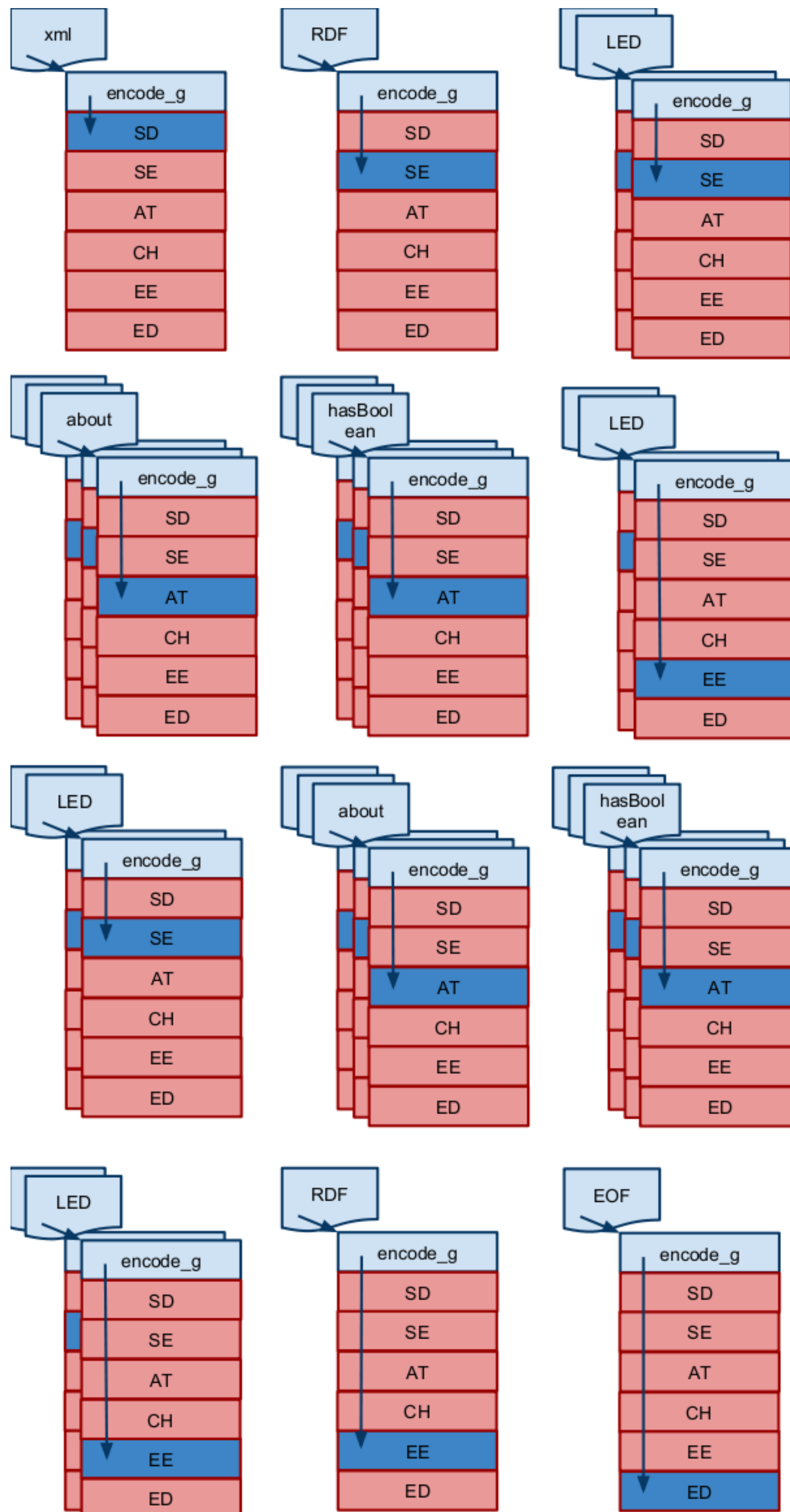

Figura 5.8: Esempio di funzionamento dello stack di grammatiche

l'occupazione del segmento codice all'interno della memoria *ROM* dei nodi sensori.

Allo stato attuale queste funzioni consentono di popolare la memoria per schemi che, come quelli visti finora, hanno fino a 2 livelli di annidamento. Benchè come già sottolineato il formato consenta di esprimere anche complessitá maggiori.

InitData

È la funzione che inizializza la memoria, consente cioè di inserire quello che viene definito `mainel`, l'elemento principale dello schema, che ha la funzione di contenere gli altri.

Nella prima parte della memoria viene infatti sempre scritta la struct dell'elemento principale, che contiene i puntatori agli altri punti della memoria dove sono allocati gli altri elementi del documento XML.

Poichè la memoria è un segmento consecutivo di byte, che viene preallocato dall'utilizzatore della libreria, uno dei parametri della funzione InitData è un vettore di interi che contiene tante celle quanti sono i campi della struct principale del documento, e per ogni campo il numero massimo di entry che ci potranno essere. Viene quindi specificato a priori, per ogni elemento del documento XML quante sono le sue occorrenze. In questo modo tra le destinazioni puntate dai campi della struct del `mainel` si mantiene spazio sufficiente per scrivere il contenuto di ogni elemento.

Il prototipo della funzione è il seguente:

```
int InitData(xchar * occ, struct schema_struct * schema,
int num, void * mem_ptr, int mem_size );
```

Riprendendo l'esempio usato in questo capitolo il vettore di occupazioni per lo schema LED, che genererà la memoria vista in figura 5.3 sarà del tipo `xchar occ={2};` ovvero 1 solo elemento con 2 occorrenze.

In particolare dopo la chiamata alla funzione InitData, la memoria sarà popolata come in figura 5.9

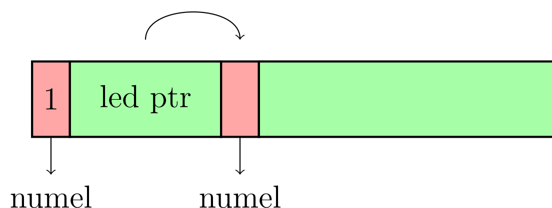

Figura 5.9: Memoria dopo la fase di Init

SetData

Dopo aver popolato la parte iniziale della memoria è possibile inserire i dati nello spazio preparato dalla funzione InitData.

Come gia spiegato i dati del documento XML vengono contenuti in struct, queste struct possono essere poi copiate nel punto opportuno della memoria inizializzata.

La funzione SetData ha questo scopo:

```
int SetData(int datatype, void * data, void * mem_ptr, struct
schema_struct * schema);
```

Essa riceve a parametro la struct contenente i dati, il puntatore alla memoria dove ricopiare la struct e un indice detto `datatype` che è l'indice nella struct che corrisponde al `mainel` dell'elemento che si vuole inserire.

La funzione si occupa quindi di inserire l'elemento nel punto opportuno della memoria, e di aggiornare il contatore `numel` che indica quanti elementi sono presenti per ogni tipo.

Read Data

Con questa funzione è possibile fare l'operazione inversa. Specificando infatti l'indice che corrisponde alla posizione nella struct `mainel` dell'elemento che si vuole leggere, la funzione restituisce il primo elemento di quel tipo. Ad ogni invocazione successiva la funzione restituirà un altro elemento come in una operazione di *pop* da una pila.

5.4.3 Le funzioni di codifica/decodifica dei tipi

Sono state sviluppate funzioni di codifica e decodifica per i tipi di dato EXI attualmente supportati, evidenziati in tabella 4.3.

Senza scendere nel dettaglio del loro funzionamento si sottolinea solamente che anche in questo caso il puntatore passato esplicitamente è quello dell'area di memoria, mentre il puntatore al buffer EXI è sempre il parametro globale.

Si noti inoltre che tutti i tipi implementati rispettano la specifica EXI, per la codifica delle stringhe si è però applicata una opzione più restrittiva: i parametri del codificatore *valuePartitionCapacity* e *valueMaxLength* sono settati a 0, questo vuol dire che le stringhe sono codificate sempre per intero anche se sono ripetute all'interno del documento XML.

La specifica EXI prevede invece il supporto ad una codifica ridotta in questi casi, codifica ridotta che però richiede la creazione *run-time* nel codificatore EXI di n tabelle di stringhe, una per ogni *exi element*. Il motivo di

questo elemento nella specifica risiede nel fatto che lavorando in nella modalità *schema not informed* è necessario codificare ogni attributo come stringa, visto che non se ne conosce il tipo, ed ugualmente, è necessario codificare i nomi degli element.

Si è quindi valutato che la complessità a livello di codice introdotta da questa funzionalità fosse superiore alla sua utilità considerando infatti la scarsa possibilità di incontrare stringhe ripetute visto che il codificatore implementato lavora sempre in modalità ***schema informed***.

5.5 Ottimizzazioni

Si da ora una panoramica delle ottimizzazioni introdotte nel codice sviluppato. Svariati sono infatti gli accorgimenti che si sono trovati durante lo sviluppo della tesi per ridurre il volume del segmento dati e del segmento codice della libEXI.

Spesso le tecniche usate non vanno nella direzione di quelle proposte in letteratura per l'ottimizzazione del codice, questo perchè il problema della dimensione del codice viene raramente considerato e si tende a concentrarsi maggiormente sulle prestazioni. Come si vedrà nelle conclusioni, capitolo 6 esse si sono comunque rivelate pienamente soddisfacenti.

5.5.1 Grammatiche

Il formato delle grammatiche presentato in 5.3.4 contiene, usando semplici valori interi, gran parte dell'informazione che costituisce una grammatica.

Si noti che il tipo di questi array è `xchar`, un tipo definito nel file `document.h` che può essere impostato a `unsigned char` per la maggior parte degli utilizzi. Ogni grammatica può quindi contenere al più 255 produzioni, valore sufficiente per tutti gli schemi del progetto SENSEI, e per la maggior parte degli schemi presi in esame anche durante la fase di test. In altro caso basta cambiare la definizione del tipo xchar ad un tipo di dato che consenta di memorizzare valori maggiori.

Il preprocessore ottimizza inoltre le grammatiche a livello di schema, eliminando le grammatiche che hanno le stesse produzioni pur essendo state generate a partire da elementi differenti. L'eliminazione è trasparente poichè le grammatiche sono puntate solo dall'*array All grammar* visto in 5.2.2.

In secondo luogo il preprocessore "fonde" anche le grammatiche per pur appartenendo ad elementi presenti in schemi differenti sono tra loro uguali.

Con queste tecniche si elimina molta della ridondanza che schemi simili tra loro, o che contengano elementi identici come struttura, introducono.

5.5.2 Parametri

La scelta di rendere globali i parametri che definiscono lo schema in uso durante l'operazione di codifica o il buffer contenente lo stream EXI, deriva dall'osservazione sperimentale sulla riduzione sensibile dello spazio occupato sia nel segmento codice che nello stack, dalla libreria.

Si è infatti verificato come il passaggio di molti parametri, benchè per riferimento, arrivasse ad avere una influenza considerevole. Ad esempio: la riduzione da 3 a 1 parametro per le funzioni di codifica/decodifica dei tipi ha portato ad un risparmio di circa 500byte.

Sempre prestando attenzione a questi dettagli si è notato come spesso la tecnica di tradurre piccoli pezzi di codice ripetuti molte volte in funzioni, anzichè ridurre la dimensione del codice compilato la aumentava. L'osservazione del disassemblato del codice ha permesso di notare che le istruzioni richieste per fare il *jump* incondizionato alla funzione, e la copia dei parametri che essa avrebbe dovuto ricevere pesavano effettivamente di più del codice completo inserito, benchè ripetuto, mediante il sistema delle MACRO offerte dal C.

5.5.3 Analisi del codice

L'analisi del codice macchina, mediante l'uso di un disassemblatore *C*, ha inoltre consentito di scoprire ulteriori possibili ottimizzazioni, come l'utilizzo di cicli `do while` per ridurre il numero di istruzioni macchina. In altri casi si sono invece notati segmenti di istruzioni tra loro identiche che potevano essere posizionati in altri punti dell'interprete di automi per consentirne l'esecuzione eliminandone la ripetizione.

5.5.4 Attivatori di funzione

Le funzioni di codifica e decodifica dei tipi di dato e le funzioni che implementano la pila di automi sono racchiuse dentro direttive per il compilatore *C* di tipo `#ifdef`.

In questo modo il preprocessore, durante l'elaborazione degli schemi può produrre delle direttive `#define` che in fase dei compilazione del processore *C* stabiliscono quali funzioni "attivare" per una certa istanza del processore. Così facendo solo le funzioni di codifica/decodifica dei tipi di dato usati dalle classi di documenti XML che ci si aspetta di processare vengono incluse nella versione compilata del codice consentendo di ridurre ulteriormente la dimensione del codice prodotto.

Capitolo 6

Risultati e Conclusioni

Si presentano ora i risultati ottenuti dalla tesi, come visto nel capitolo 5 l'implementazione realizzata copre solo parzialmente la specifica EXI [9], tuttavia la libreria realizzata soddisfa i requisiti di progetto ed offre ampie prospettive di estensione.

Si è infatti realizzata una implementazione estremamente efficiente sia in termini di prestazioni che di ingombro in memoria. Il codice è multiarchitettura e facilmente includibile all'interno di progetti che necessitino di un codificatore EXI.

I test comparativi sono stati effettuati contro l'unica implementazione EXI liberamente disponibile, *EXIeficient* [8], libreria originariamente sviluppata da *Siemens* [21], e recentemente rilasciata come *software libero*.

I test di occupazione di memoria sono stati effettuati usando il compilatore `msp430-gcc` [22] e le misurazioni sono state fatte sul codice compilato per i nodi.

Gli schemi XSD e i documenti XML usati sono disponibili in appendice.

6.1 Test di memoria

Uno degli obbiettivi principale di progetto è stato quello di produrre una libreria efficiente ma che mantenga un ingombro minimo in memoria.

Si analizzano ora i risultati raggiunti in questo senso.

6.1.1 Occupazione Nodo

Si vede in figura 6.1 e 6.2ed in tabella 6.1 la situazione di occupazione della memoria all'interno di un nodo che abbia caricato tutto il codice del progetto

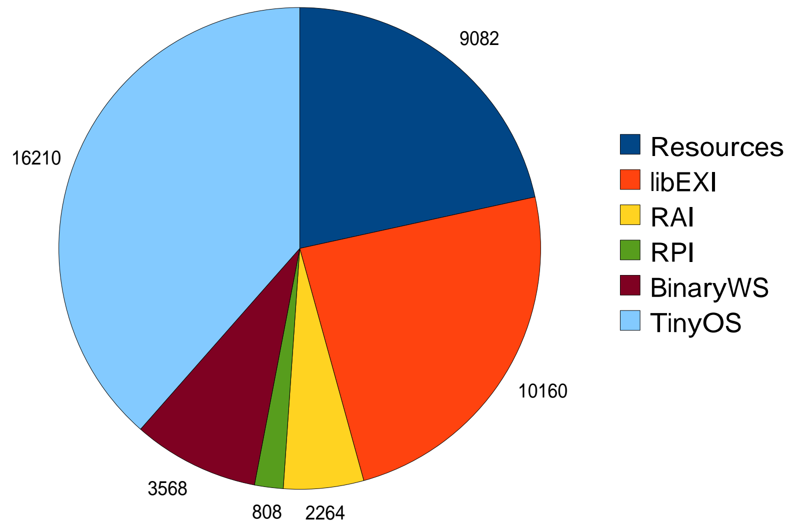

Figura 6.1: Occupazione totale della memoria ROM del nodo Tmote Sky (memoria totale 48KByte)

SENSEI i valori di utilizzo della libreria fanno riferimento alla parte di EXI compreso l'utilizzo delle API.

6.1.2 Occupazione libreria

Se si analizza nel dettaglio la libreria (figura 6.3 e tabella 6.2) si può vedere come l'occupazione della libreria sia principalmente nelle funzioni di codifica e decodifica dei dati.

Il valore chiamato `Inizializzazione` si riferisce all'occupazione dei file header contenenti le definizioni delle *struct* ausiliare necessarie al funzionamento del processore.

Il valore `Grammatiche` fa invece riferimento al peso in memoria delle grammatiche relative agli schemi. La misura è relativa alle grammatiche generate dagli schemi del progetto SENSEI *Actuators.xsd, Sensors.xsd, Subscription.xsd* disponibili ai listati A.1,A.2 e A.3 in appendice A.

Il valore `API` indica invece quanti bytes occupano le API viste in 5.4.3 mentre `Utilizzo API` fa riferimento al costo di utilizzo di queste API al-

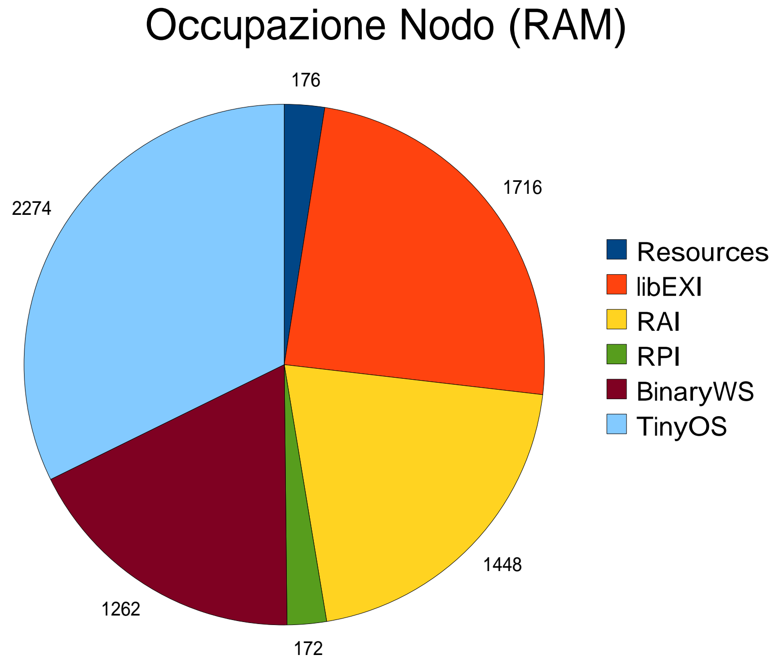

Figura 6.2: Occupazione totale della memoria RAM del nodo Tmote Sky (memoria totale 8KByte)

Tabella 6.1: Occupazione totale del nodo (bytes)

Componenti EXI	ROM	RAM
Resources	9082	176
libEXI	10160	1716
RAI	2264	1448
RPI	808	172
BinaryWS	3568	1262
TinyOS	16210	2274
Totale	42092	7048

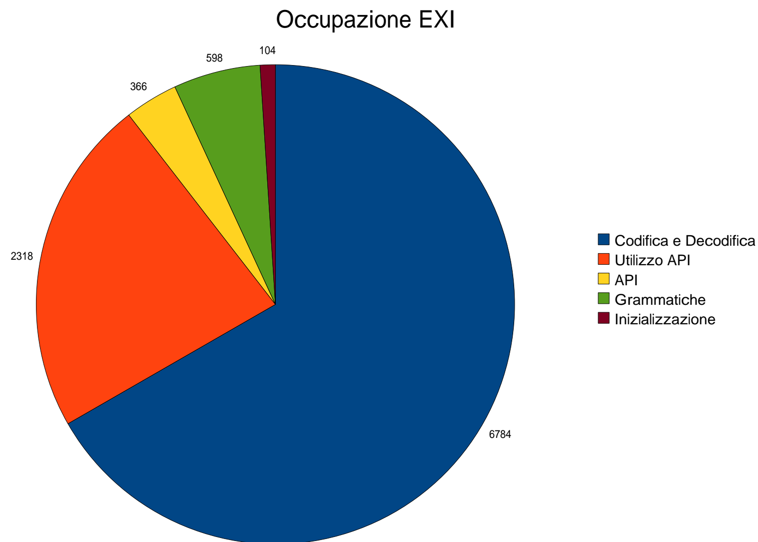

Figura 6.3: Occupazione della libEXI all'interno del nodo (bytes)

Tabella 6.2: Occupazione della libEXI all'interno del Nodo (bytes)

Componenti libEXI	ROM	RAM
Codifica e Decodifica	6784	98
API	366	0
Utilizzo API	2318	24
Grammatiche	598	590
Inizializzazione	104	1006

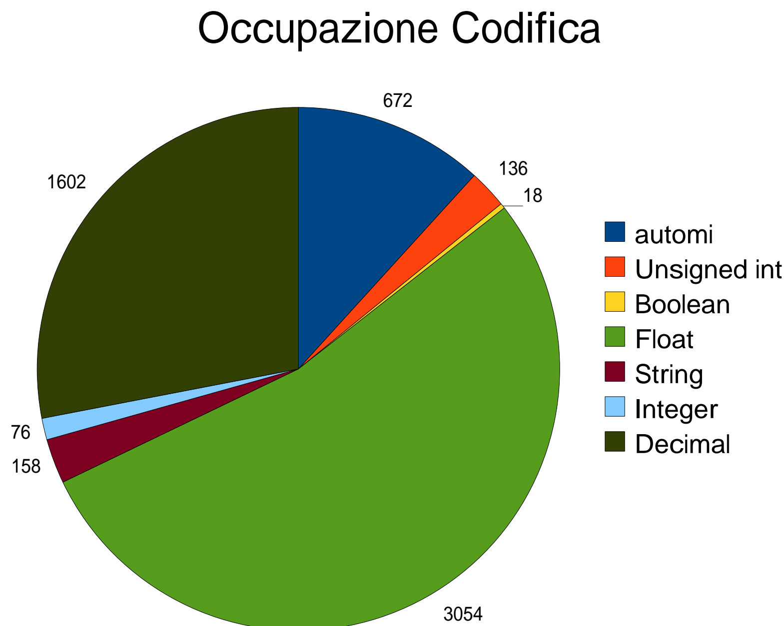

Figura 6.4: Occupazione delle funzioni di codifica(bytes)

l'interno del codice del progetto SENSEI. Si vede dunque che, benché esse siano un ottimo strumento per il programmatore nel facilitare l'utilizzo della libreria, è conveniente che il sistema di acquisizione e salvataggio del codice che si avvale della *libEXI* sia pensato per scrivere i dati nel formato di input definito in 5.2.1, consentendo così un notevole risparmio di memoria.

Infine, il valore `codifica/decodifica` raccoglie il costo in memoria di tutte le funzioni di codifica e decodifica il cui dettaglio viene ora analizzato.

6.1.3 Occupazione funzioni di codifica e decodifica

In figura 6.4 ed in tabella 6.3 si analizzano le occupazioni delle singole funzioni di codifica implementate dalla libEXI

Si può notare che il costo maggiore sia costituito dalle funzioni `Float` e `Decimal`, questo perchè il loro funzionamento richiede l'inclusione delle funzioni `expf` e `logf` della `lib math` del *C*.

In figura 6.5 ed in tabella 6.4 si analizzano le occupazioni delle singole funzioni di decodifica implementate dalla libEXI

Tabella 6.3: Occupazione delle funzioni di codifica (bytes)

Funzioni di codifica	ROM	RAM
Automa	672	76
UInt	136	0
Boolean	18	0
Float	3054	0
String	158	0
Integer	76	0
Decimal	1602	0
Somma delle singole	5716	76
Totale	4662	76

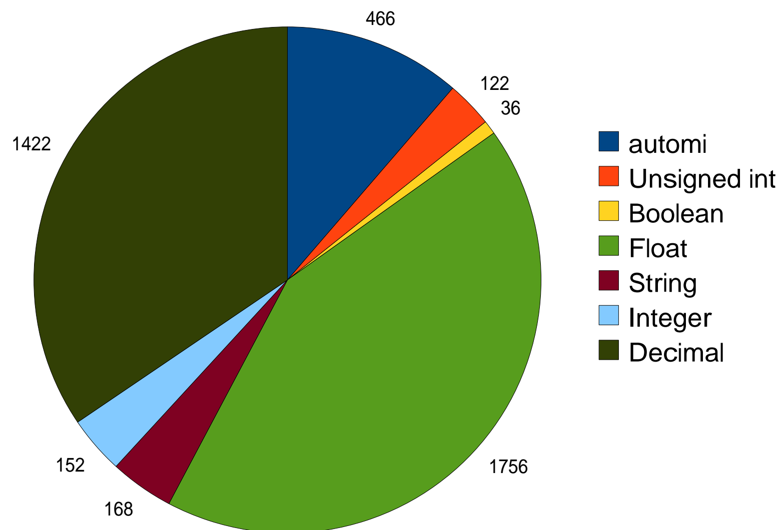

Figura 6.5: Occupazione delle funzioni di decodifica (bytes)

Tabella 6.4: Occupazione delle funzioni di decodifica (bytes)

Funzioni di decodifica	ROM	RAM
Automa	466	52
UInt	122	0
Boolean	36	0
Float	1756	0
String	168	0
Integer	152	0
Decimal	1422	0
Somma delle singole	4122	52
Totale	3134	52

Anche in questo caso si vede come il maggior costo sia dovuto alle funzioni `Float` e `Decimal` che come le loro duali usano funzioni che richiedono la `lib math`.

I pesi chiamati `encode automa` e `decode automa` si riferiscono al costo delle funzioni `codifica`,`encode_g_seq` e `decodifica`, `decode_g_seq`. Il nucleo della libEXI pesa quindi poco più di 1000 byte.

Si noti infine che i valori qui elencati sono stati calcolati attivando singolarmente le singole funzioni di codifica e decodifica, sono quindi da interpretarsi come valori "assoluti". I costi calcolati sono anche "puliti" da eventuali costi dovuti a eventuali dipendenze di una funzione su altre.

In tabella 6.3 e 6.4 si può infatti vedere come, in caso di inclusione contemporanea di tutte le funzioni di codifica e decodifica all'interno del codice compilato, il peso totale sia minore della somma dei pesi delle singole funzioni. Questo perchè il compilatore è in grado di effettuare ottimizzazioni che eliminano eventuali duplicazioni di istruzioni, in particolare l'inclusione delle funzioni della libreria matematica viene a pesare meno, in valore assoluto, sul codice finale

Il superiore costo delle funzioni di codifica rispetto a quelle di decodifica è certamente legato alla diversa complessità delle operazioni effettuate ed al maggior numero di funzioni coinvolte nell'operazione. La codifica di un tipo complesso richiede infatti l'utilizzo di funzioni di codifica di tipi più "semplici". Inoltre la codifica di un dato richiede una chiamata di scrittura per ogni *byte* di pacchetto codificato mentre nella decodifica avviene l'opposto.Il fatto che le operazioni di scrittura richiedano un maggior numero di istruzioni macchina unito al fatto che la decodifica di un singolo tipo avviene tutta all'interno della stessa funzione giustifica quindi queste differenze di dimensione.

6.2 Prestazioni

Le prestazioni temporali della libreria sono state misurate testandone il funzionamento su un PC con il seguente *hardware*:

- CPU: `Intel(R) Core(TM)2 CPU 6300 @ 1.86GHz`
- Memoria Ram: `1GB DDR2`
- Sistema Operativo: `Ubuntu Linux 10.4`
- Versione Kernel: `2.6.27-17-generic`

Le misurazioni sono state effettuate solo sulla parte della libreria chiamata *processore*, la parte che si occupa cioè della codifica e decodifica del pacchetto EXI.

I primi test sono stati effettuati usando gli schemi del progetto SENSEI e i documento XML che vengono codificati normalmente. Questi documenti di esempio si possono vedere in appendice B ai listati B.2,B.1 e B.3 e fanno riferimento agli schemi visti in A.

Data la ridotta dimensione dei documenti, il pacchetto EXI ad essi corrispondente è tra i 7 e i 30 bytes, per poter ottenere misurazioni significative si sono dovuti effettuare *run* multipli e calcolare i tempi medi.

In figura 6.6 vediamo dunque come il tempo medio si stabilizzi dopo un certo numero di interazioni e si attesti su un valore di tra i 7 e i 2 micro secondi (dipendentemente dalla complessità del documento).

In figura 6.7 vediamo invece i tempi totali calcolati durante lo stesso test, si nota che la crescita e' praticamente lineare con il numero di iterazioni.

Per verificare questo aspetto si è dunque scelto di produrre un documento XML la cui taglia partisse da 10 elementi sino a raggiungere i 900000, lo schema di riferimento per questo documento è stato A.1. Ne vediamo un estratto al listato 6.1.

Listato 6.1: XML documento di taglia elevata

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mas:Mass xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:xs="
    http://www.w3.org/2001/XMLSchema" xmlns:mas="urn:sensei:rai">
  <mas:LED mas:about="1" mas:hasBooleanValue="0" />
  <mas:LED mas:about="1" mas:hasBooleanValue="0" />
  <mas:LED mas:about="1" mas:hasBooleanValue="0" />
  <mas:LED mas:about="1" mas:hasBooleanValue="0" />
  ...
  ...
  <mas:LED mas:about="1" mas:hasBooleanValue="0" />
  <mas:LED mas:about="1" mas:hasBooleanValue="0" />
  <mas:LED mas:about="1" mas:hasBooleanValue="0" />
  <mas:LED mas:about="1" mas:hasBooleanValue="0" />
```

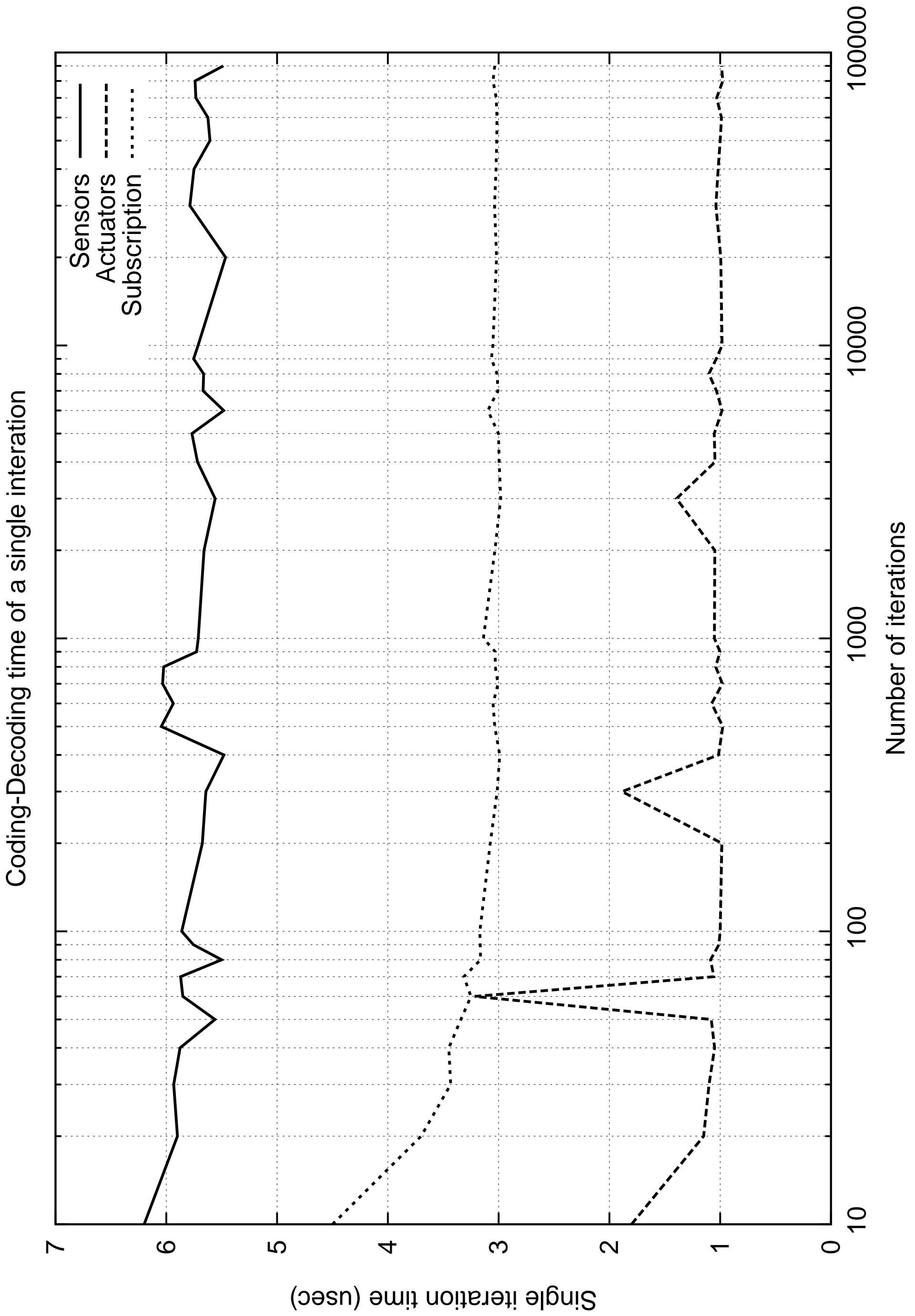

Figura 6.6: Grafico prestazioni su xml SENSEI

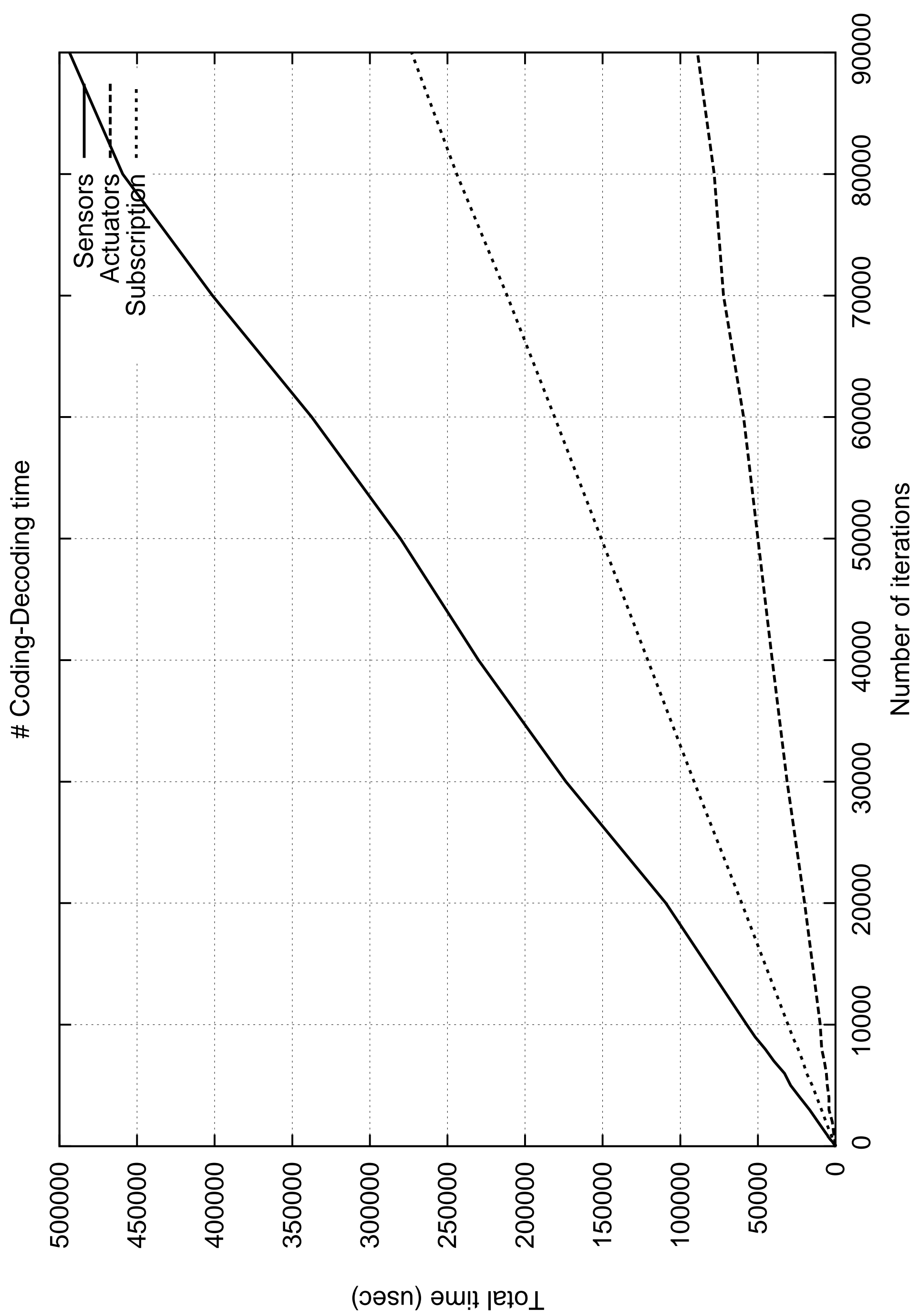

Figura 6.7: Grafico prestazioni su xml SENSEI

```
  <mas:LED mas:about="1" mas:hasBooleanValue="0" />
  <mas:LED mas:about="1" mas:hasBooleanValue="0" />
</mas:Mass>
```

I risultati ottenuti e visibili nelle figura 6.8 mostrano come i tempi crescano linearmente con la taglia del documento XML che viene codificato, il grafico 6.9 (in scala logaritmica) mostra i tempi per l'iterazione di un singolo elemento del documento. I tempi diminuiscono e si stabilizzano perchè i costi delle operazioni di misurazione diventano trascurabili.

Sono stati inoltre effettuati dei test comparativi con l'unica altra implementazione di EXI liberamente disponibile: ***EXIeficient*** [8]. EXIeficient è una libreria sviluppata in *java* che implementa quasi totalmente la specifica EXI, lo sviluppo è stato iniziato da *Siemens* che ha poi rilasciato il codice sotto licenza *GPL* ma continua comunque a mantenere degli sviluppatori interni.

I test comparativi sono stati effettuati usando le stesse impostazioni per l'header EXI e lavorando nella sola modalità *schema informed*.

Le misure sono state effettuate sulla sola parte di codifica e decodifica, senza considerare i tempi di generazione delle grammatiche.

Essi non sono infatti confrontabili in quanto il *preprocessore Ruby* utilizzato nella libEXI implementa solo una parte dello standard ed effettua delle operazioni di ottimizzazione *intra-schema* ed *inter-schema* che non sono effettuate nell'implementazione di *EXIeficient*.

EXIeficient si è inoltre rivelata fondamentale durante lo sviluppo della tesi per avere un ulteriore metodo di paragone dei risultati ottenuti (oltre a quelli previsti dalla teoria).

Nel grafico 6.10 si vede quindi il risultato di questi test. Le prestazioni del codice *C* sono sensibilmente migliori ma si vede comunque la linearità dei tempi anche nella implementazione *java*.

6.3 Performance EXI

Le *performance* di EXI come standard sono ovviamente prestabilite, ma è comunque interessante vedere il rapporto di compressione ottenuto nei documenti più usati all'interno del progetto SENSEI.

- lo schema B.2 passa da 242 bytes a 10 con un rapporto di compressione di 24/1
- B.1 633 bytes a 58, con un rapporto di 10/1
- B.3 passa da 225 bytes a 38, con un rapporto di 5/1

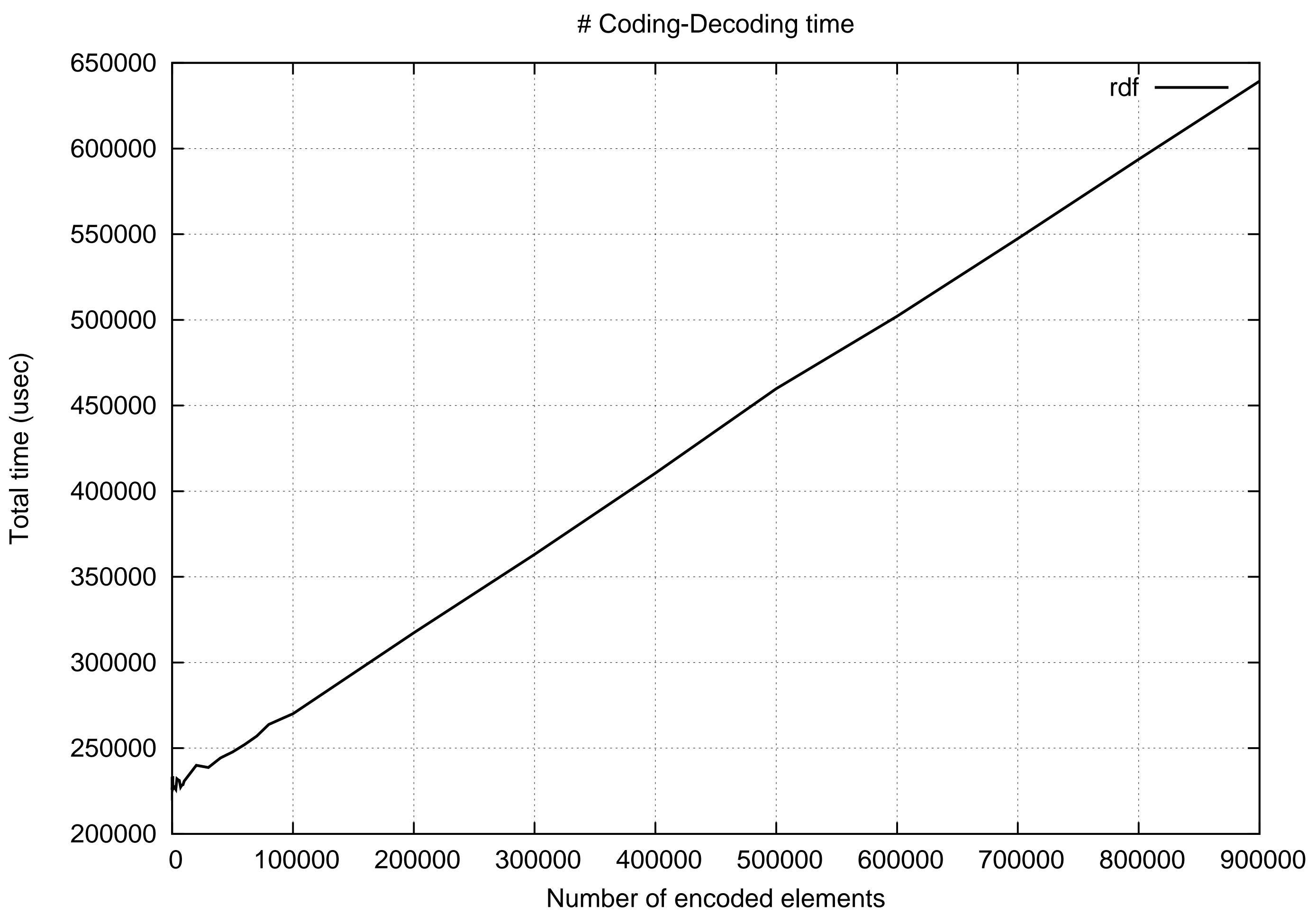

Figura 6.8: Grafico prestazioni su documento a dimensione variabile

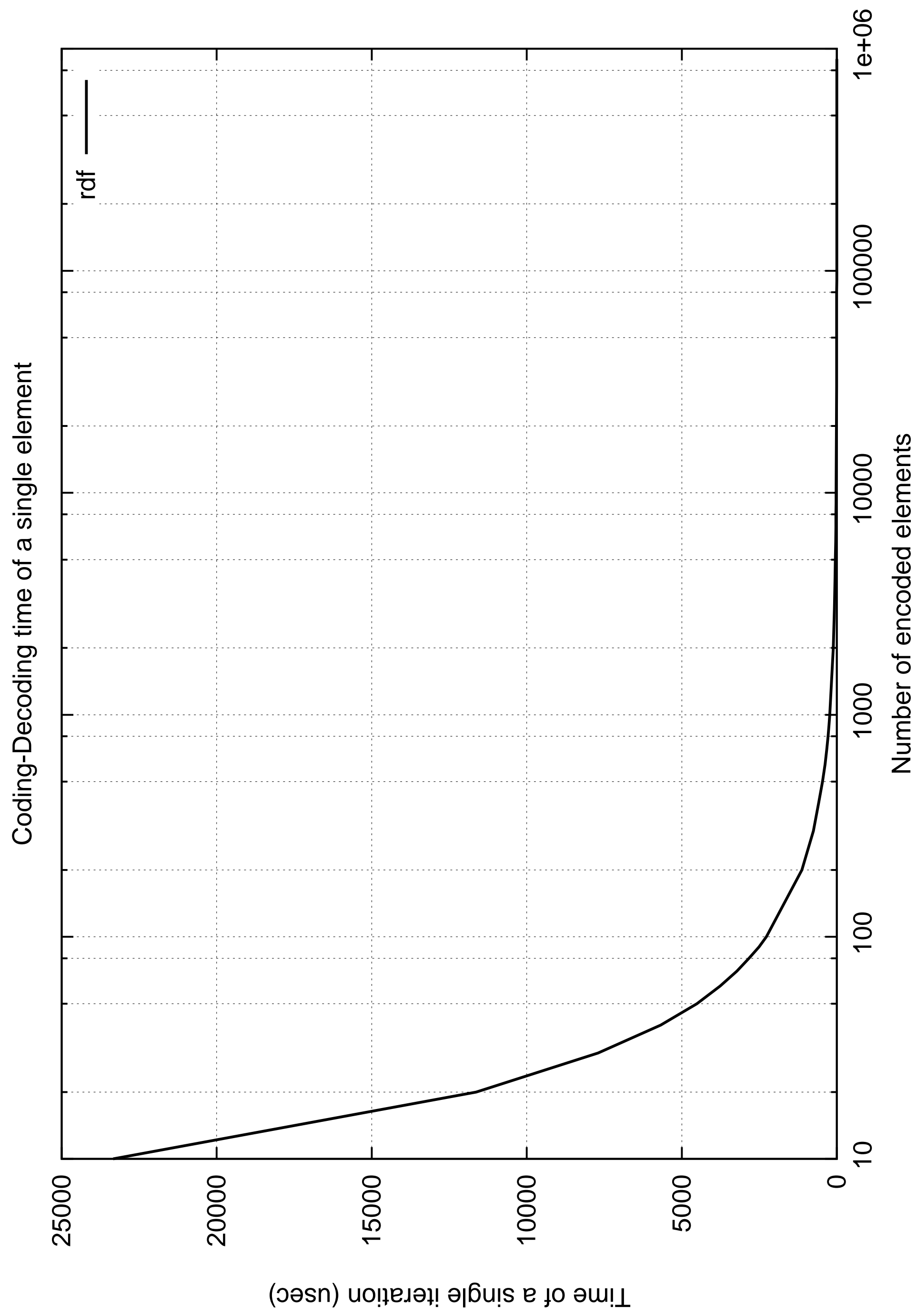

Figura 6.9: Grafico prestazioni su documento a dimensione variabile

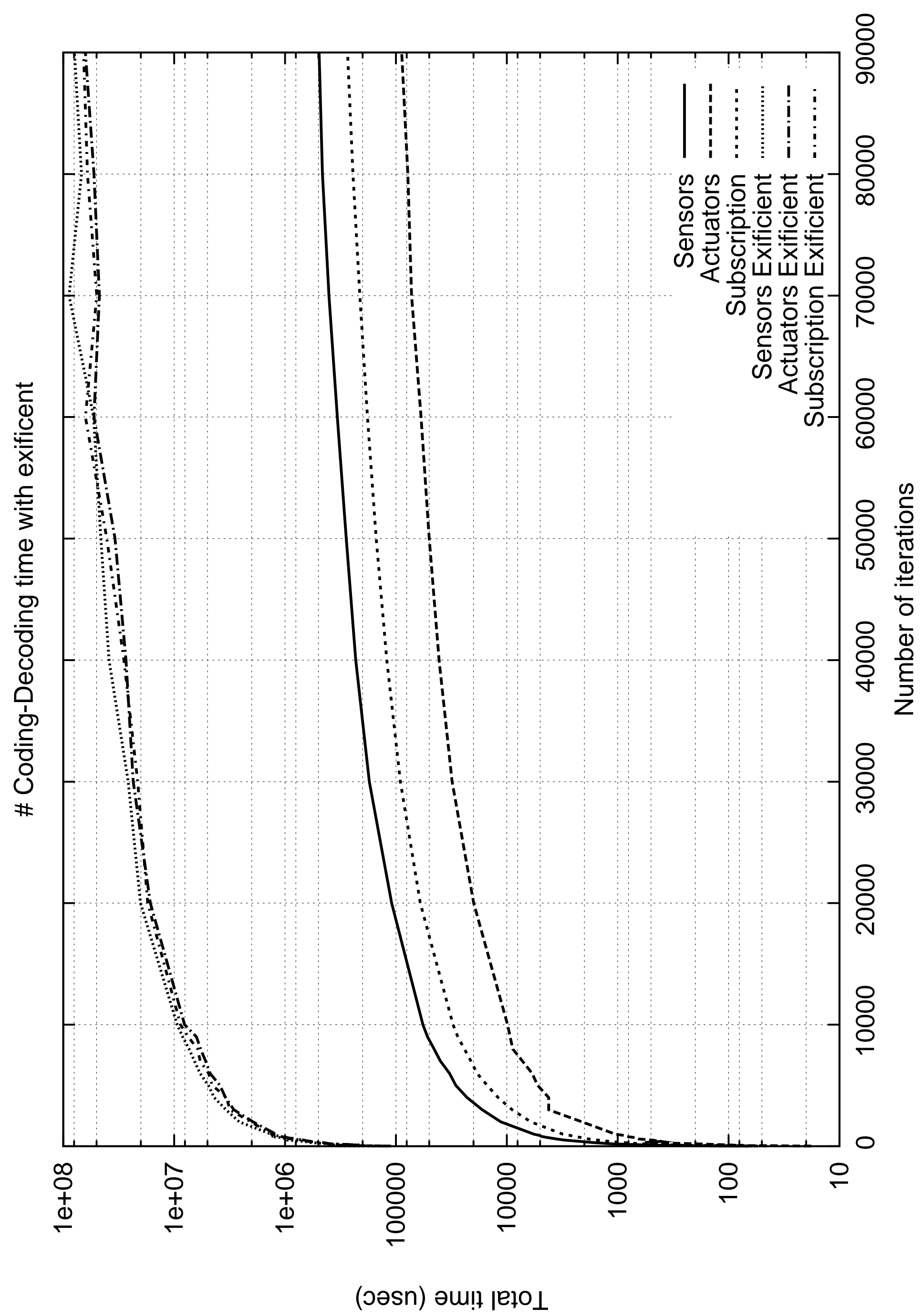

Figura 6.10: Grafico prestazioni su xml SENSEI, libEXI Vs EXIeficient

6.4 Conclusioni

Il lavoro di tesi ha quindi prodotto una libreria multipiattaforma che implementa una parte consistente dello standard EXI ed è in grado di effettuare una codifica/decodifica efficiente sia in termini di prestazioni che di dimensione del codice.

La scelta di lavorare nella sola modalità *schema informed* si è rivelata il punto chiave della tesi ed ha portato a produrre del codice estremamente ottimizzato che ha consentito l'inserimento della libreria all'interno di un progetto come SENSEI.

La scelta di un formato intermedio tra XML e EXI per la codifica dei dati ha consentito la loro gestione anche all'interno dei nodi eliminando gran parte dell'*overhead* rispetto a quello che si sarebbe avuto con una decodifica completa. Il costo delle API in termini di risorse puó essere ulteriormente ridotto dal programmatore che usi la libEXI e istruisca il suo codice affinchè i dati vengano direttamente memorizzati durante la loro elaborazione nel formato previsto dalla libEXI.

Un altro importante risultato raggiunto riguarda il formato di memorizzazione delle grammatiche, la struttura utilizzata ha consentito di limitare al massimo l'ingombro di questi dati complessi utilizzando solamente *array* di tipo `char`.

La principale direzione verso la quale si possono concentrare gli sviluppi futuri di questa tesi riguarda il *preprocessore*. L'implementazione qui esposta ha fatto infatti uso del linguaggio *ruby* principalmente per far fronte alla complessa natura del problema della generazione delle grammatiche. Certamente nell'ipotesi di integrare anche il preprocessore in maniera più consistente all'interno di un progetto una sua riscrittura in linguaggio *C* è auspicabile.

Un'ulteriore miglioramento preso in considerazione riguarda la possibilità per i nodi sensori di effettuare il caricamento dinamico delle grammatiche dalla memoria *flash* di cui sono dotati per consentire al nodo di tenere in memoria le informazioni necessarie alla codifica di molti schemi.

L'aggiunta del supporto ad altri tipi di dato è già prevista nella versione attuale della libEXI e l'operazione sarebbe principalmente costituita dalla scrittura delle funzioni di codifica e decodifica dei dati.

Infine si sottolinea che l'inserimento della modalità *schema non-informed* non è al momento in esame e questa opzione richiederebbe una totale ripianificazione del codice della libreria a causa della diversa natura del problema da risolvere. Il lavoro di tesi ha evidenziato che questa parte dello standard EXI è difficilmente implementabile su dispositivi che abbiano risorse *hardware* paragonabili a quelle dei nodi sensori.

Appendice A

Schemi XSD

A.1 Schemi progetto SENSEI

Listato A.1: Schemi Actuators

```
file Actuators.xsd
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="
    qualified" targetNamespace="urn:sensei:rai" xmlns:rdf="http://www.w3.org
    /1999/02/22-rdf-syntax-ns#" xmlns:res="urn:sensei:rai">
  <xs:import namespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      schemaLocation="ActuatorsRDF.xsd"/>

  <xs:element name="LED">
    <xs:complexType>
      <xs:attribute ref="rdf:about" use="required"/>
      <xs:attribute name="hasBooleanValue" use="required" form="qualified"
          type="xs:boolean"/>
    </xs:complexType>
  </xs:element>


</xs:schema>

file ActuatorsRDF.xsd
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="
    qualified" targetNamespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:res="
    urn:sensei:rai">
  <xs:import namespace="urn:sensei:rai" schemaLocation="Actuators.xsd"/>
  <xs:element name="RDF">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="res:LED" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:attribute name="about" type="xs:nonNegativeInteger"/>
</xs:schema>
```

Listato A.2: Schemi Sensors

```
file Sensors.xsd
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="
    qualified" targetNamespace="urn:sensei:rai" xmlns:rdf="http://www.w3.org
    /1999/02/22-rdf-syntax-ns#" xmlns:res="urn:sensei:rai">
  <xs:import namespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      schemaLocation="SensorsRDF.xsd"/>

  <xs:element name="Temperature">
    <xs:complexType>
      <xs:attribute ref="rdf:about" use="required"/>
      <xs:attribute name="hasDecimalValue" use="required" form="qualified"
          type="xs:decimal"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Light">
    <xs:complexType>
      <xs:attribute ref="rdf:about" use="required"/>
      <xs:attribute name="hasDecimalValue" use="required" form="qualified"
          type="xs:decimal"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="LightInfraRed">
    <xs:complexType>
      <xs:attribute ref="rdf:about" use="required"/>
      <xs:attribute name="hasDecimalValue" use="required" form="qualified"
          type="xs:decimal"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Button">
    <xs:complexType>
      <xs:attribute ref="rdf:about" use="required"/>
      <xs:attribute name="hasBooleanValue" use="required" form="qualified"
          type="xs:boolean"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="HumidityRelative">
    <xs:complexType>
      <xs:attribute ref="rdf:about" use="required"/>
      <xs:attribute name="hasDecimalValue" use="required" form="qualified"
          type="xs:decimal"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="ADC">
    <xs:complexType>
      <xs:attribute ref="rdf:about" use="required"/>
      <xs:attribute name="hasDecimalValue" use="required" form="qualified"
          type="xs:decimal"/>
    </xs:complexType>
  </xs:element>

    <xs:element name="Pressure">
    <xs:complexType>
      <xs:attribute ref="rdf:about" use="required"/>
      <xs:attribute name="hasDecimalValue" use="required" form="qualified"
```

```
          type="xs:decimal"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Position">
    <xs:complexType>
      <xs:attribute ref="rdf:about" use="required"/>
      <xs:attribute name="hasXvalue" use="required" form="qualified" type="
          xs:decimal"/>
      <xs:attribute name="hasYvalue" use="required" form="qualified" type="
          xs:decimal"/>
      <xs:attribute name="hasZvalue" use="optional" form="qualified" type="
          xs:decimal"/>
    </xs:complexType>
  </xs:element>

</xs:schema>

file SensorsRDF.xsd
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="
    qualified" targetNamespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:res="
    urn:sensei:rai">
  <xs:import namespace="urn:sensei:rai" schemaLocation="Sensors.xsd"/>
  <xs:element name="RDF">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="res:Temperature" minOccurs="0" maxOccurs="unbounded
            "/>
        <xs:element ref="res:Position" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="res:Light" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="res:LightInfraRed" minOccurs="0" maxOccurs="
            unbounded"/>
        <xs:element ref="res:Button" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="res:HumidityRelative" minOccurs="0" maxOccurs="
            unbounded"/>
        <xs:element ref="res:ADC" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="res:Pressure" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:attribute name="about" type="xs:nonNegativeInteger"/>
</xs:schema>
```

Listato A.3: Schemi Subscription

```
file Subscription.xsd
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="Threshold">
    <xs:complexType>
      <xs:sequence>
        <!-- Operation type -->
        <xs:element name="Operation" minOccurs="1" maxOccurs="1">
          <xs:simpleType>
            <xs:restriction base="xs:decimal">
              <xs:enumeration value="0" /> <!-- Equal to -->
              <xs:enumeration value="1" /> <!-- Greater than -->
              <xs:enumeration value="2" /> <!-- Less than -->
```

```
            </xs:restriction>
          </xs:simpleType>
       </xs:element>
        <!-- Threshold value. Type can be e.g. int or decimal  -->
        <xs:element name="Value" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Subscription">
    <xs:complexType>
      <xs:sequence>
        <!-- URL to send the event data to. If only a path, then the source
            IP address of the requester is assumed. If no URL is given, then
             the default notification path is assumed.  -->
        <xs:element name="URL" type="xs:anyURI" minOccurs="0" maxOccurs="1"/
            >
        <!-- The subscription ID generated by the subscriber, which must be
            included in the corresponding notifications if given. -->
        <xs:element name="ID" type="xs:integer" minOccurs="0" maxOccurs="1"/
            >
        <!-- Request an event over a specific threshol (optional) -->
        <xs:element ref="Threshold" minOccurs="0" maxOccurs="1"/>
        <!-- The period of the event in ms (optional) -->
        <xs:element name="Period" type="xs:double" minOccurs="0" maxOccurs="
            1"/>
        <!-- If this element exists, only send a new event if the value has
            changed -->
        <xs:element name="OnlyOnChange" type="xs:boolean"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Appendice B

Documenti XML

B.1 Documenti XML SENSEI

Listato B.1: Documento di esempio Sensors

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:xs="
    http://www.w3.org/2001/XMLSchema" xmlns:res="urn:sensei:rai">
  <res:Temperature rdf:about="2" res:hasDecimalValue="22.500000"/>
  <res:Light rdf:about="2" res:hasDecimalValue="22.500000"/>
  <res:LightInfraRed rdf:about="1" res:hasDecimalValue="29.500000"/>
  <res:Button rdf:about="5" res:hasBooleanValue="6"/>
  <res:HumidityRelative rdf:about="1" res:hasDecimalValue="29.500000"/>
  <res:ADC rdf:about="1" res:hasDecimalValue="29.500000"/>
  <res:Pressure rdf:about="1" res:hasDecimalValue="29.500000"/>
</rdf:RDF>
```

Listato B.2: Documento di esempio Actuators

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf2:RDF xmlns:rdf2="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:xs=
    "http://www.w3.org/2001/XMLSchema" xmlns:res="urn:sensei:rai">
  <res:LED rdf2:about="2" res:hasBooleanValue="6"/>
</rdf2:RDF>
```

Listato B.3: Documento di esempio Subscription

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sub:Subscription xmlns:sub="subscription">
        <sub:URL>http://test</sub:URL>
        <sub:ID>1</sub:ID>
        <sub:Period>5.5</sub:Period>
        <sub:OnlyOnChange>1</sub:OnlyOnChange>
</sub:Subscription>
```

Bibliografia

[1] I. Bekmezci, “Wireless sensor networks: A military monitoring application.”

[2] Y. S. E. Cayirci, “Wireless sensor networks: a survey.” Computer Networks 38, 2002.

[3] Moteiv Corporation, “Tmote Sky: Datasheet.” datasheet.

[4] Crossbow Rechnology, “TelosB Mote Platform.” datasheet.

[5] Brian W. Kernighan; Dennis M. Ritchie, *The C Programming Language (2nd ed.)*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[6] Francesco Fornasiero, “Interfacciamento di una rete di sensori tramite tecnologia 6LowPAN.” Master Thesis, 2009.

[7] M. Harvan, “Connection Wireless Sensor Networks to the Internet - a 6lowpan Implementation fo TinyOS 2.0.” Master Thesis, 2007.

Siti web consultati

[8] "Exificient." http://exificient.sourceforge.net/.

[9] W3C, "Efficient xml interchange (exi) format 1.0," 2009. http://www.w3.org/TR/2009/CR-exi-20091208/.

[10] W3C, "Efficient xml interchange (exi) primer," 2009. http://www.w3.org/TR/exi-primer/.

[11] W3C, "World wide web consortium." http://www.w3.org/.

[12] S. Project, "Sensei project," 2007. http://www.ict-sensei.org/.

[13] B. University, "Berkley university." http://www.berkley.edu/.

[14] C. Technology, "Crossbow technology." http://www.xbow.com/.

[15] Contiki, "The operating system for embedded smart objects,the internet of things." http://www.sics.se/contiki.

[16] TinyOS, "Tinyos." http://www.tinyos.net.

[17] "A distributed environment for prototyping ad hoc networks." http://www.btnode.ethz.ch.

[18] "Multimodal networks of in-situ sensors." http://mantis.cs.colorado.edu.

[19] "A wireless sensor networking real-time operating system." http://www.nanork.org.

[20] "Embedded operating system." https://projects.nesl.ucla.edu/public/sos-2x.

[21] "Siemens r&d." http://www.ct.siemens.com/.

[22] "Msp430 gcc." http://mspgcc.sourceforge.net/.

[23] "Ruby language." http://ruby-lang.org/.

[24] "Xml language." http://www.w3.org/XML/.

[25] "Ipv6 specification." http://www.ipv6.org/.

Elenco delle tabelle

4.1 opzioni header EXI 25
4.2 Eventi EXI supportati 26
4.3 Datatypes EXI (in grigio quelli implementati) 31

6.1 Occupazione totale del nodo (bytes) 71
6.2 Occupazione della libEXI all'interno del Nodo (bytes) 72
6.3 Occupazione delle funzioni di codifica (bytes) 74
6.4 Occupazione delle funzioni di decodifica (bytes) 75

Elenco delle figure

1.1 Uso di EXI e XML per le comunicazioni in reti di dispositivi eterogenei 3
1.2 Schema del progetto SENSEI 3
1.3 Scenario globale del sistema 4

2.1 Wireless Sensors Networks 8
2.2 Architettura tiny node 8
2.3 Il nodo TMote Sky visto dal basso 10
2.4 Il nodo TMote Sky visto dall'alto 10

3.1 Componenti e Interfacce in NesC 15
3.2 Scenario globale del sistema 17
3.3 Architettura a livelli del sistema 18

4.1 Automa associato ad una semplice grammatica regolare 23
4.2 struttura pacchetto EXI 24
4.3 Grammatiche come automi 28
4.4 Stack di grammatiche riferito listato a 4.4 e tratta da [10] . . . 35
4.5 Document Grammar 40
4.6 RDF element Grammar 41
4.7 Led Grammar 42
4.8 Esempio di codifica EXI 42

5.1 Schema di funzionamento 46
5.2 Schema comunicazione della memoria 48
5.3 Rappresentazione in memoria del documento XML 49
5.4 Schema di funzionamento del Preprocessore 54
5.5 L'oggetto schema ActuatorsRDF 56
5.6 Struttura di un oggetto Proto Grammar 56
5.7 Struttura del processore EXI 59
5.8 Esempio di funzionamento dello stack di grammatiche 63
5.9 Memoria dopo la fase di Init 64

6.1 Occupazione totale della memoria ROM del nodo Tmote Sky (memoria totale 48KB
6.2 Occupazione totale della memoria RAM del nodo Tmote Sky (memoria totale 8KB
6.3 Occupazione della libEXI all'interno del nodo (bytes) 72
6.4 Occupazione delle funzioni di codifica(bytes) 73
6.5 Occupazione delle funzioni di decodifica (bytes) 74
6.6 Grafico prestazioni su xml SENSEI 77
6.7 Grafico prestazioni su xml SENSEI 78
6.8 Grafico prestazioni su documento a dimensione variabile . . . 80
6.9 Grafico prestazioni su documento a dimensione variabile . . . 81
6.10 Grafico prestazioni su xml SENSEI, libEXI Vs EXIeficient . . 82

Elenco dei listati

4.1 Esempio di grammatica EXI 27
4.2 Esempio di grammatica per l'elemento pluto 27
4.3 Esempio di grammatica EXI 32
4.4 Schema XML esempio 34
4.5 Grammatica EXI non normalizzata 36
4.6 Grammatica di un Elemento EXI 36
4.7 Grammatica EXI normalizzata con event codes 37
4.8 Schema di tipo xsd Actuators.rdf 38
4.9 Schema di tipo xsd ActuatorsRDF.rdf 38
4.10 Documento XML da codificare 39
4.11 Documento XML da codificare 39
5.1 Schema di tipo XSD ActuatorsRDF.rdf 48
5.2 Struct corrispondenti allo schema precedente 49
5.3 Documento XML da codificare 49
5.4 Documento XML da codificare 50
5.5 Grammatiche relative allo schema LED 51
5.6 Grammatiche C relative allo schema LED 51
6.1 XML documento di taglia elevata 76
A.1 Schemi Actuators 85
A.2 Schemi Sensors 86
A.3 Schemi Subscription 87
B.1 Documento di esempio Sensors 89
B.2 Documento di esempio Actuators 89
B.3 Documento di esempio Subscription 89

Ringraziamenti

Ringrazio te che mi hai accompagnato.
Non importa se per il tempo di una sigaretta, per mesi o per anni interi.
Ti ringrazio perchè mi hai lasciato qualcosa.
Ti ringrazio perchè mi hai educato.
Ti ringrazio perchè mi hai insegnato.
Ti ringrazio perchè mi sei stato amico.
Ti ringrazio perchè mi sei stato accanto.
Ti ringrazio perchè mi hai fatto ridere.
Ti ringrazio perchè mi hai amato. [1]

[1] Dico davvero, grazie.