View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Thesis and Dissertation Archive - Università di Pisa

UNIVERSITA' DEGLI STUDI DI PISA
Facoltà di Scienze Matematiche, Fisiche e Naturali
Dipartimento di Informatica

La domotica e Internet. Una soluzione per l'interoperabilità

Tesi di laurea in Domotica

Relatore: Prof. Rolando Bianchi Bandinelli
Correlatore: Dott. Vittorio Miori

Tesi di Laurea di:
Dario Russo, matricola 195031

Anno Accademico 2005-2006

Ai miei genitori, ai miei nonni, ad Ada.
A tutti quelli che ho conosciuto durante questa avventura universitaria e con me hanno condiviso vittorie e delusioni.

Sommario

La presente relazione illustra gli obiettivi ed i risultati ottenuti durante la mia collaborazione con il Laboratorio di Domotica presso l'Istituto di Scienza e Tecnologie dell'Informazione ISTI "Alessandro Faedo" del CNR di Pisa, nell'ambito del progetto di ricerca *HATS* (Home Automation Technologies and Standards)[1].

Lo scopo della tesi è quello di proporre una soluzione praticabile che possa garantire la definitiva affermazione della domotica.

Tale soluzione consiste nella realizzazione di un *framework* per internet *open source* e rilasciato sotto licenza *GNU*, basato sulle tecnologie standard dei *Web Service* e su una *grammatica XML*, detta *domoML*, che permetta l'integrazione e l'interoperabilità dei servizi offerti dai *middleware* domotici attualmente presenti sul mercato.

[1]http://hats.isti.cnr.it

Ringraziamenti

Ringrazio il dottor Rolando Bianchi Bandinelli per avermi dato l'opportunità di lavorare ad un'importante progetto di ricerca, il dottor Vittorio Miori per il costante supporto tecnico.

Un sentito riconoscimento al dottor Marcello Ferri, per la sua infinita disponibilità nei miei riguardi senza il quale i tempi di realizzazione del prototipo sarebbero stati sensibilmente più lunghi, e a chiunque abbia sostenuto il mio lavoro, anche solo con un consiglio o un incoraggiamento.

Indice

Sommario **I**

Ringraziamenti **III**

1 Introduzione **2**

1.1 Inquadramento generale 3

1.2 Struttura della tesi 5

2 La domotica allo stato dell'arte **6**

2.1 I middleware domotici 9

2.1.1 *EIB*, *BatiBus* e *EHS* 10

2.1.2 *Konnex* 22

2.1.3 *UPnP 1.0* (*Universal plug and play*) . . . 49

3 I Web Service **54**

3.1 Definizione 55

3.2 Componenti dell'architettura 58

3.2.1 Entità 58

3.2.2 Operazioni 59

3.2.3 Artefatti 60

3.3 Ciclo di vita 60
3.4 *Stack* protocollare 61
3.5 Protocolli specifici 66
3.5.1 *SOAP* 66
3.5.2 *WSDL* (*web service Description Language*) 72
3.5.3 *WSDL Specification* 72
3.5.4 *UDDI* (*Universal Description Discovery and Integration*) 75

4 Impostazione del problema 78
4.1 Scenario 79
4.2 Un primo sguardo all'architettura domoNet . . . 83
4.3 Terminologia 86

5 DomoML 87
5.1 *DomoDevice* 88
5.2 *DomoMessage* 94

6 Lato server: il web service 97
6.1 Eseguire un *domoMessage* 98
6.2 La cooperazione tra differenti *tech manager* . . . 99
6.3 Il *tech manager* 100
6.3.1 Esecuzione di un *domoMessage* 101

7 Lato client: il domoNetClient 104
7.1 Interazione tra il *domoNetClient* ed il *domoNet-ClientUI* 105

8 Il prototipo **107**
8.1 Strumenti di lavoro 107
8.2 Lo sviluppo 109
8.2.1 Il *domoML* 111
8.2.2 Il *web service* 120
8.2.3 Il *DomoNetClient* 135
8.3 Il test 139
8.3.1 Interruttore *Konnex* con lampada *UPnP* . 140
8.3.2 Interruttore *UPnP* con lampada *Konnex* . 143

9 Conclusioni **147**
9.1 Realizzazioni sperimentali e valutazioni 147
9.2 Integrazione con Internet 148
9.3 Direzioni future di ricerca 149

A Versione *alpha* del manuale del prototipo **151**
A.1 Introduction 151
A.1.1 The domotics 153
A.1.2 The solution: *DomoNet* 156
A.1.3 Terminology 158
A.1.4 How to install, compile and launch the application 159
A.2 DomoML 164
A.2.1 What *domoML* is and what it does 164
A.3 Server side 166
A.3.1 Setting up the *webService* 166

A.3.2 The *techManagers* 170
A.4 Client side 181

Bibliografia **184**

Elenco delle figure **188**

Elenco delle tabelle **190**

Capitolo 1

Introduzione

La presente relazione illustra gli obiettivi ed i risultati ottenuti durante la mia collaborazione con il Laboratorio di Domotica presso l'Istituto di Scienza e Tecnologie dell'Informazione ISTI "Alessandro Faedo" del CNR di Pisa, nell'ambito del progetto di ricerca *HATS* (Home Automation Technologies and Standards)[1]. Il progetto *HATS* nasce per sviluppare un'architettura basata su un insieme di nuove tecnologie, al fine di creare ambienti domestici intelligenti che consentano l'integrazione e l'interoperabilità tra i servizi offerti.

Un aspetto fondamentale per valutare la qualità di un ambiente domestico intelligente, risiede nella capacità di scoprire, configurare e controllare i dispositivi ed i servizi eterogenei che lo caratterizzano; pertanto, le parole chiave per il successo del progetto *HATS* sono integrazione ed interoperabilità. La cosiddetta *HAN* (Home Area Network), su cui si basa il modello

[1] http://hats.isti.cnr.it

proposto da *HATS*, è rappresentata dal framework domoNet. domoNet segue i princìpi tipici di un'architettura orientata ai servizi (SOA - Service Oriented Architecture) fondata su quella dei recenti *Web Service*. Proprio questa similitudine ha suggerito l'impiego dei *Web Service* e delle tecnologie standard ad essi correlate, quali *XML*, come base per la realizzazione del framework domoNet.

1.1 Inquadramento generale

Il gergo delle nuove tecnologie si è andato arricchendo di termini quali *Home Automation*, *Building Automation*, *Smart Home*, etc., tutti sinonimi derivati da un altro neologismo, *Domotique*, coniato in Francia e composto dalla fusione dei due termini *Domus* e *Informatique*. Tale neologismo, ripreso in Italia con il termine *Domotica*, rappresenta l'applicazione delle tecnologie di Informazione e Comunicazione al mondo domestico, allo scopo di migliorarne il comfort ed il controllo.

In questo inesplorato panorama di sviluppo, l'industria ed il mercato hanno giocato, e giocano tuttora, un ruolo di primaria importanza per il definitivo decollo delle tecnologie domotiche. Da un lato i finanziamenti provenienti dall'industria hanno proposto numerosi *middleware*, standard sempre più sofisticati; dall'altro la presenza di troppi standard scarsamente interoperabili, ognuno dei quali promosso da coalizioni aziendali diverse, ha reso la domotica un ordigno inesploso.

Il "boom" che il mondo della ricerca e quello dell'industria si aspettavano non è di fatto ancora avvenuto: l'impiego reale della tecnologia domotica è tuttora ostacolato dal tentativo di ciascuna industria di imporre il proprio standard sugli altri. La presenza di un così vasto numero di standard domotici, oramai ampiamente affermati, indica che difficilmente vi sarà la definitiva consacrazione di uno solo di essi. La torta da spartirsi è molto invitante, e nessuno vuol rinunciare alla propria fetta. D'altra parte, è altrettanto improbabile che tutte le coalizioni che promuovono le proprie scelte, si riuniscano intorno ad un tavolo per realizzare un unico *middleware* che rappresenti lo standard *de facto* per la domotica.

La nostra proposta è dunque quella di una soluzione praticabile che possa garantire la definitiva affermazione della domotica. Tale soluzione consiste nella realizzazione di un framework, basato sulle tecnologie standard dei *Web Service* e su una *grammatica XML*, detta *domoML*, che permetta l'integrazione e l'interoperabilità dei servizi offerti dai *middleware* attualmente presenti sul mercato. Per ovvie ragioni, il modello proposto rappresenta solo una parte dell'intero framework; in particolare dimostreremo come il prototipo realizzato garantisca l'interoperabilità tra due dei più importanti, e tra loro diversi, *middleware domotici*: *Konnex*[2] e *UPnP*[3].

[2]http://www.konnex.org

[3]http://www.upnp.org

1.2 Struttura della tesi

Delineiamo dapprima una panoramica sulla domotica e sulle numerose tecnologie ad essa correlate, con particolare enfasi sui due *middleware* oggetto del prototipo: *Konnex* e *UPnP*.

Evidenzieremo poi le similitudini tra l'architettura dei *Web Service* e quella proposta dal progetto *HATS* per lo sviluppo del framework, descrivendo le tecnologie necessarie su cui si basa l'implementazione.

Infine, descriveremo ampiamente la fase operativa: dallo schema architetturale del *framework domoNet*, alla formalizzazione di una *grammatica XML standard* per la comunicazione tra applicazioni domotiche all'interno del *framework* (*domoML*).

Capitolo 2

La domotica allo stato dell'arte

La domotica costituisce l'insieme delle tecnologie che si occupano dell'automazione degli edifici, e in primo luogo dell'integrazione dei dispositivi elettronici, degli elettrodomestici, dei sistemi di comunicazione e controllo, ivi presenti.

Gli scopi che le applicazioni domotiche si prefiggono possono essere così suddivisi:

- incremento del livello di comfort:
  le applicazioni domotiche devono rendere più accogliente lo spazio abitativo, facilitandone la gestione soprattutto per gli utenti disabili (ad es. standard per l'audio/video, per il controllo e il monitoraggio remoto di elettrodomestici, luci, tapparelle, porte, etc);

- raggiungimento di un adeguato livello di sicurezza:
  i sistemi domotici devono garantire l'incolumità dell'utente a fronte di situazioni di emergenza (ad es. tecniche di antiintrusione, rilevamento di incendi, fughe di gas, etc);

- ricerca di tecniche per il risparmio energetico:
  l'impiego della domotica deve consentire una gestione più accurata ed efficiente del livello dei consumi energetici (ad es. strumenti avanzati per ottenere informazioni che permettano una più equa distribuzione dei carichi energetici).

È indubbio quindi che la domotica proponga scenari avveniristici, ed è altrettanto certo l'impatto positivo che il suo impiego potrebbe avere sulla qualità della nostra vita.

Proprio per questo motivo, gli ultimi anni sono stati il teatro per lo sviluppo di soluzioni, promosse soprattutto da aziende private, tese ad accaparrarsi tutti i benefici derivanti da un nuovo e presumibilmente fiorente mercato.

Tuttavia, ad oggi, non si percepisce un livello di sensibilità e di conoscenza tale da permettere il decollo, ed il definitivo consolidamento, di un mercato dedicato alla domotica.

Paradossalmente, le tecnologie e gli standard domotici diversi, e tra loro scarsamente interoperabili, sono talmente numerosi da rappresentare un ostacolo all'espansione del mercato.

Dal punto di vista dell'utente finale diventa molto complicato, e forse anche poco comprensibile, percepire la necessità di acquistare prodotti domotici rispetto a quelli tradizionali. Infatti i potenziali benefici che deriverebbero dall'acquisto di questi prodotti non riescono a giustificare uno sforzo economico maggiore, per quanto l'introduzione di intelligenza, intesa come capacità

di calcolo, all'interno di un qualsiasi dispositivo, comporti costi aggiuntivi sempre più esigui.

Inoltre, a causa della scarsa interoperabilità tra i vari standard, per sfruttare a pieno i benefici della domotica, l'utente sarebbe costretto ad acquistare solo i prodotti conformi ad un particolare sistema. Ciò potrebbe verificarsi solo in due condizioni: l'acquisto contemporaneo di tutti i dispositivi presenti nell'edificio, oppure una cognizione tecnica che permettesse la conoscenza dello standard utilizzato per poter acquistare ulteriori dispositivi conformi ad esso.

Entrambe le condizioni sono però di difficile realizzazione, dal momento che nella maggior parte dei casi, l'ambiente domestico è molto dinamico: la topologia e la dislocazione dei suoi elementi cambiano frequentemente e i dispositivi vengono acquistati in momenti diversi. Si pensi, ad esempio, agli elettrodomestici: è quantomeno raro che tutti quelli presenti in una abitazione siano acquistati contemporaneamente, e comunque si tratta di oggetti che nel tempo finiscono con il deteriorarsi, usurarsi e diventare obsoleti.

Per comprendere meglio queste limitazioni è di aiuto un esempio classico di applicazione domotica: si supponga di avere una caffettiera che possa essere accesa tramite una radiosveglia ad un'ora prestabilita. Se la radiosveglia e la caffettiera parlano la stessa lingua, ossia sono entrambe conformi al medesimo standard, ogni mattina sarà possibile avere il caffè pronto appena

svegli. Tuttavia, se per qualsiasi motivo, uno dei due dispositivi deve essere cambiato, sarà necessario rimpiazzarlo con un altro conforme allo stesso standard.

Sarebbe invece auspicabile permettere all'utente di scegliere i dispositivi indipendentemente dallo standard cui appartengono: in questo modo l'utente non dovrebbe minimamente conoscere le specifiche tecniche del proprio sistema, ma potrebbe esclusivamente trarne tutti i possibili benefici.

Lo sforzo di quella parte della comunità scientifica che si occupa di domotica è teso, dunque, a infondere la necessaria sensibilizzazione affinché si affermi definitivamente l'impiego di queste tecnologie.

2.1 I middleware domotici

I middleware domotici si possono dividere in due grandi classi che differiscono sostanzialmente per il sistema di comunicazione usato: una, più tradizionale, fa uso di un sistema *bus*; l'altra, più recente, si basa su un insieme di protocolli più evoluto come *TCP/IP*. Del primo tipo fanno parte standard oramai consolidati nel mondo della domotica, quali ad esempio *X10*, *EIB*, *BatiBus*, *EHS*, *Konnex*, *LonWorks*, *CEBus*; la seconda tipologia invece include, tra gli altri, standard come *UPnP*, *Jini*, *OSGi*.

A fianco di un particolare sistema di comunicazione, è disponibile un numero sempre maggiore di standard che si basa-

no su mezzi trasmissivi diversi, come *IEEE 802.11b* (*Wi-Fi*), *Bluetooth*, *IEEE 1394* (*Firewire*), *IrDA*, *ZigBee*, etc.

Andiamo ora ad analizzare i principali standard appartenenti alle due classi di middleware: *EIB*, *BatiBus*, *EHS* e *Konnex* per quanto riguarda i sistemi *bus*, e *UPnP* per quelli basati su *TCP/IP*.

2.1.1 *EIB*, *BatiBus* e *EHS*

Il sistema *bus*

Per controllare tutte le apparecchiature della casa occorrono numerose connessioni cablate e parecchi telecomandi: aggiungerne di nuovi può portare ad una situazione difficile da gestire.

In una normale casa moderna, infatti, possono coesistere numerose reti di diversa natura:

- rete telefonica;
- rete elettrica;
- antenna per la ricezione del segnale televisivo terrestre;
- antenna per la ricezione del segnale televisivo satellitare;
- rete di calcolatori locale (LAN);
- rete idrica, etc.

È necessario quindi escogitare un sistema per ridurre la complessità dei vari canali di comunicazione presenti.

Il sistema *bus* rappresenta una potenziale soluzione al problema del cablaggio in un ambiente domestico: esso costituisce l'unico mezzo di comunicazione al quale sono collegati tutti i dispositivi in parallelo e su cui transitano tutte le informazioni.

Nello specifico caso di un ambiente domestico, i dispositivi vengono alimentati da una linea di alimentazione dedicata e comunicano attraverso il *bus*.

È possibile collegare al *bus* diverse categorie di dispositivi, come ad esempio sensori e attuatori, elettrodomestici, impianti audio/video.

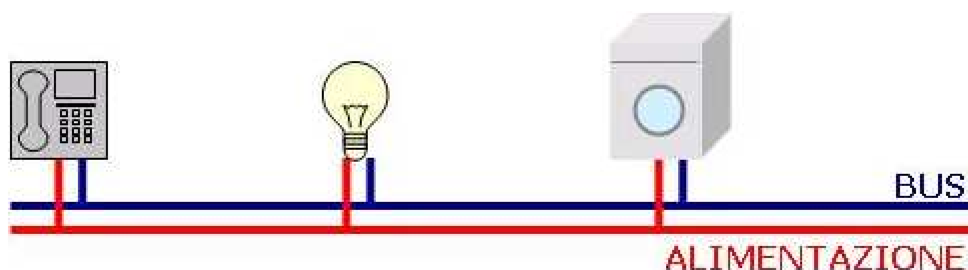

Figura 2.1: Esempio di sistema bus.

I vantaggi dell'utilizzo di questo sistema sono di immediata comprensione: il tradizionale cablaggio *punto-a-punto* è sostituito da un unico "cavo" che percorre il volume dell'edificio. Grazie a ciò è possibile eseguire dei collegamenti tra i dispositivi in modo del tutto dinamico, cioè senza installare cablaggi aggiuntivi.

Affinché il sistema *bus* risulti efficace, è necessario però che tutti i dispositivi che vi sono collegati comunichino usando la stessa lingua, ovvero che siano conformi allo stesso standard.

Inoltre, la filosofia di interazione tra i dispositivi deve seguire un unico schema logico (configurazione manuale del sistema, *plug and play*, etc.).

All'interno del *bus* vengono trasmessi dei piccoli pacchetti, detti anche frame o telegrammi, che possono essere distinti in base al tipo di informazione che trasportano:

- comandi da eseguire;
- segnali di allarme;
- segnali per il riconoscimento dei dispositivi;
- segnali di conferma dell'esecuzione di azioni (ACK);
- segnali che notificano lo stato dei dispositivi, etc.

Per poter collegare un dispositivo al *bus* è necessaria un'interfaccia, costituita da un componente elettronico detto *BCU* (*Bus* Coupling Unit). Il *BCU* interpreta i pacchetti ricevuti dal *bus*, li traduce nei segnali adatti al dispositivo, e viceversa. Ogni *BCU* possiede un indirizzo, non necessariamente univoco, che lo identifica; il meccanismo di indirizzamento e di trasmissione differisce a seconda degli standard.

Il sistema *bus* non specifica un unico mezzo trasmissivo: malgrado alcuni testi riportino il doppìno di rame come mezzo trasmissivo standard, abbiamo ritenuto opportuno elencare tutti i mezzi attualmente disponibili sul mercato:

1. Onde convogliate (PL = Power Line):

   Con questo mezzo trasmissivo si utilizzano i cavi della rete che distribuisce l'alimentazione elettrica (corrente alternata di 220 volt a 50 Hz) alla quale si aggiunge un basso voltaggio modulato (da 3 a 148 KHz) che non influisce in modo significativo sulla potenza distribuita.

   Poiché le onde convogliate viaggiano nello stesso cavo che porta l'alimentazione elettrica, i *BCU* devono essere equipaggiati con un componente elettronico che separi il canale di comunicazione del *bus* da quello di alimentazione a 220 volt.

   Questo mezzo trasmissivo è molto utilizzato poiché permette di realizzare in modo semplice un sistema *bus*. Non sono necessari cablaggi aggiuntivi (tutte le case possiedono una rete elettrica), elimina l'inquinamento da onde radio e permette di raggiungere tutte le parti della casa là dove sono presenti dispositivi elettrici.

   Le frequenze di trasmissione delle onde convogliate sono state regolamentate e standardizzate dal Comitato Europeo per la Standardizzazione Elettrotecnica (*CENELEC*)[1], come mostrato in tabella 2.1:

   La maggiore limitazione è dovuta all'ampiezza di banda ammessa; le velocità che si raggiungono sono dell'ordine di qualche Kilobit per secondo, da 1,2 a 5,4 Kbps. An-

[1]http://www.cenelec.org

Frequenze (KHz)	Vincoli	Impiego
3 - 95	Accesso riservato	Fornitori di energia elettrica
95 - 125	Accesso libero	Sistemi domotici
125 - 140	Accesso regolamentato CENELEC	Sistemi domotici
140 - 148,5	Accesso riservato	Allarmi e sicurezza

Tabella 2.1: Regolamentazione sull'assegnazione delle frequenze.

che se sono teoricamente raggiungibili velocità superiori, queste risulterebbero non aderenti alla normativa; inoltre, per la natura dell'informazione trasmessa (esclusa quella audio/video), le velocità raggiunte risultano più che sufficienti.

2. Doppìno di rame (TP = Twisted Pair):

   La trasmissione è convogliata su doppìno di rame, simile al doppìno telefonico (a seconda degli standard utilizzati si possono avere caratteristiche diverse).

   Da un rapido confronto con le onde convogliate, emerge che la trasmissione su doppìno è meno soggetta ai disturbi esterni, e quindi più affidabile; la trasmissione su onde convogliate, infatti, è sottoposta ai disturbi della linea elettrica, poiché il canale di comunicazione e quello di alimentazione non sono disgiunti.

   Il doppìno può supportare velocità superiori e il costo aggiuntivo dovuto alla sua installazione non incide pesante-

mente sulla costruzione dell'edificio.

3. Cavo coassiale (CX = CoaXial cable):

   Questo mezzo trasmissivo viene usato per trasportare segnali ad alta frequenza, come ad esempio quelli dell'antenna televisiva o di quella satellitare.

   La capacità sottesa a questo utilizzo consente di instaurare comunicazioni digitali ad alta velocità, e perciò il cavo coassiale rappresenta una valida alternativa quando le prestazioni del doppìno non sono sufficienti.

   Di seguito sono descritti alcuni sistemi che, pur non utilizzando un vero e proprio cavo coassiale, possono essere collocati, per ragioni storiche, in questo contesto.

   - *FireWire*:

     creato da *Apple Computer*, è conosciuto oggi come lo standard *IEEE 1394*; è stato successivamente adottato da Sony che lo ha rinominato iLink. Usa un cavo schermato a 6 o 8 fili, ma non esattamente coassiale. Oltre alla trasmissione, i cavi FireWire forniscono anche l'alimentazione; esistono due versioni che si distinguono per la velocità di trasmissione.

   - *USB* (*Universal Serial Bus*):

     è stato introdotto per sostituire la vecchia porta seriale RS232 e la porta parallela *Centronics*. Usa un cavo schermato a 6 fili (anche questo, non propriamente

coassiale). Come per i cavi *FireWire*, anche i cavi *USB*[2] forniscono l'alimentazione ed esistono due versioni che differiscono per le velocità supportate.

La tabella 2.2 mostra il confronto tra FireWire e USB in termini di velocità di trasmissione:

Bus	rate vers.1 (Mbps)	rate vers.2 (Mbps)
FireWire	400	800
USB	12	400

Tabella 2.2: Velocità di trasmissione.

mentre la tabella 2.3 evidenzia le categorie applicative in cui trovano maggior impiego:

Bus	applicazioni vers.1	applicazioni vers.2
FireWire Audio/Video	Hard Disk esterni, etc.	Collegamenti ancora più veloci
USB	Mouse, Modem, etc.	Audio/Video, Hard Disk esterni, etc.

Tabella 2.3: Caterorie applicative.

- Onde radio (RF = Radio Frequency):
  Il *bus* a onde radio invia i comandi in radio frequenza, utilizzando le frequenze corrispondenti alla banda dei 2.4 GHz che non interferiscono con quelle utilizzate da altre apparecchiature domestiche come radio, TV, telefoni cellulari, telecomandi, etc.

[2]http://www.usb.org

Rispetto ai sistemi che utilizzano le onde convogliate o il doppìno, la trasmissione a radio frequenza è decisamente più economica anche se, in particolari condizioni di schermatura, la comunicazione può non avvenire oppure può essere molto disturbata.

Fra gli standard emergenti che utilizzano le frequenze radio della banda a 2.4GHz vi sono *IEEE 802.11b* (Wi-Fi), *Bluetooth* e, il più recente *ZigBee*.

- *IEEE 802.11b*:
  conosciuto con il nome *Wi-Fi* (*Wireless-Fidelity*)[3], rappresenta lo standard interoperabile con le reti *Ethernet* (*IEEE 802.3*).

  Raggiunge velocità nominali che si attestano sugli 11 Mbps, ma il futuro standard, denominato *IEEE 802.11a*, potrebbe raggiungere velocità dell'ordine dei 100 Mbps qualora si passi alla banda a 5 GHz.

  L'impiego di *Wi-Fi* è dedicato agli impianti altamente tecnologici, all'automazione degli elettrodomestici, alla rete locale di calcolatori, etc.;

- *Bluetooth*:
  bluetooth[4] è adeguato per trasmissioni che non superano i 10 metri di distanza, ed è ro*bus*to rispetto alle interferenze grazie alla tecnica di hopping. Questo stan-

[3]http://www.wifialliance.com
[4]http://www.bluetooth.com

dard trova impiego nella trasmissione dati e in quella vocale (telefonia fissa e mobile);

- *ZigBee*:
  rispetto a *Bluetooth*, *ZigBee*[5] amplia il raggio di trasmissione fino ad un massimo di 100 metri; come Bluetooth è molto robusto rispetto alle interferenze ma ha un consumo decisamente più basso poiché raggiunge velocità inferiori.

  ZigBee può essere impiegato come protocollo per le cosiddette reti di sensori.

4. raggi infrarossi (IR = *Infrared Rays*):
   la trasmissione via raggi infrarossi può avvenire solo tra zone che "si vedono", che sono cioè l'un l'altra raggiungibili senza dover superare ostacoli non trasparenti.

   Lo standard di riferimento è *IrDA* (*Infrared Digital Association*)[6].

   IrDA nasce nel 1994 con lo scopo di standardizzare un modello di comunicazione *punto-a-punto* per la trasmissione dei dati via raggi infrarossi; le velocità raggiungibili sono limitate a 4Mbps, mentre il raggio di azione può arrivare fino a 1 metro.

Descriviamo ora brevemente tre standard europei storicamen-

[5]http://www.zigbee.org
[6]http://www.irda.org

te molto importanti, *EIB*, *BatiBus* e *EHS*, tutti basati su un sistema *bus* ed ispirati al modello standard *ISO/OSI*.

Introduciamo quindi la struttura del modello standard *ISO / OSI*: esso prevede uno schema a 7 strati (Layer) in cui lo strato n implementa un insieme di funzionalità da esportare verso lo strato $n+1$, usando quelle esposte sull'interfaccia tra lo strato n e lo strato $n-1$.

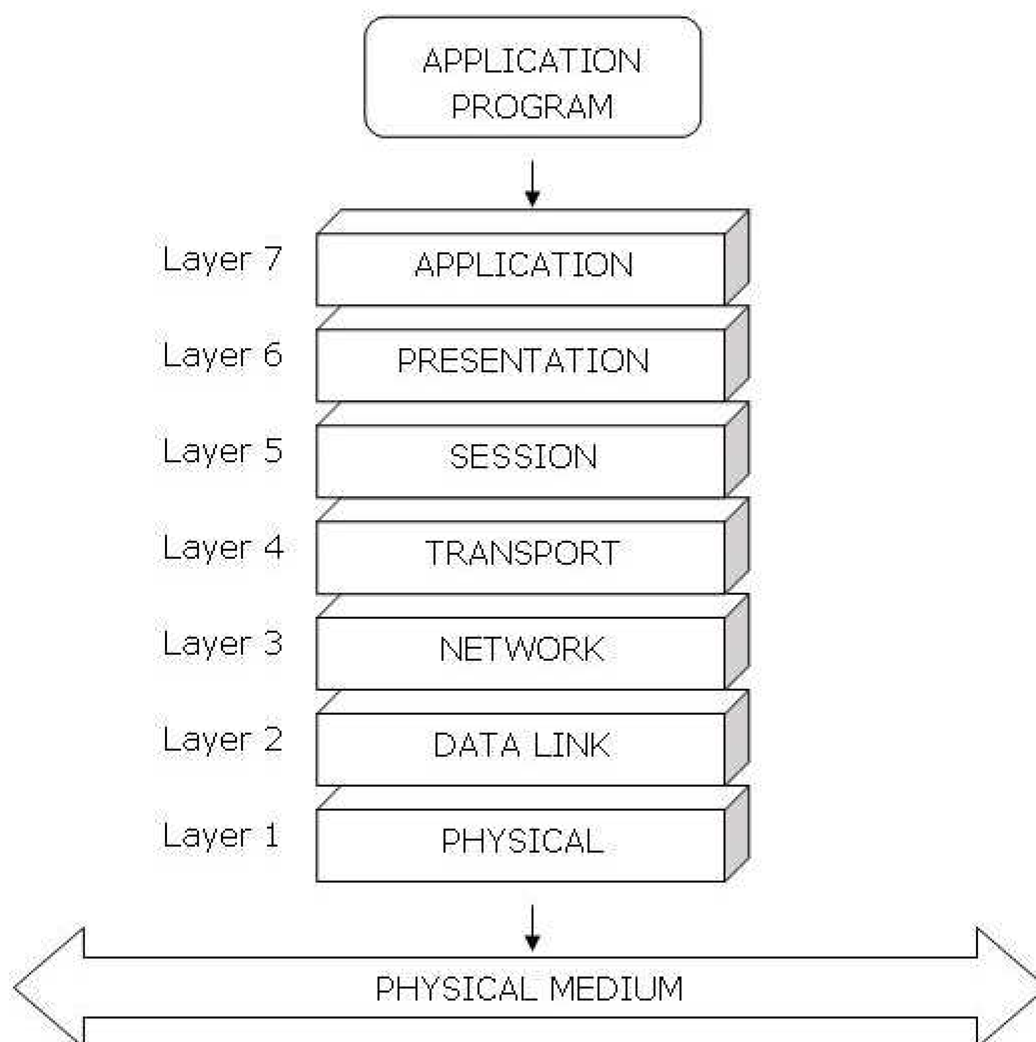

Figura 2.2: Il modello ISO / OSI.

Di seguito viene descritto brevemente ogni strato dello schema, secondo un approccio top-down:

- *application*:
  fornisce interfacce virtuali ai programmi applicativi (ad es. emulazione di terminali);

- *presentation*:
  implementa servizi per la codifica dei dati (ad es. meccanismi di compressione e crittografia);

- *session*:
  gestisce la sincronizzazione ad alto livello, controlla il dialogo e le unità di informazione;

- *transport*:
  costituisce lo spartiacque tra gli strati superiori e l'hardware degli strati inferiori: instaura tutte le connessioni di rete richieste dallo strato *session*;

- *network*:
  risolve l'indirizzamento e gestisce il traffico all'interno della rete;

- *data link*:
  riconosce e segnala agli strati superiori eventuali errori avvenuti agli strati inferiori;

- *physical*:

trasmette l'unità di informazione (bit) nei canali trasmissivi e gestisce tempi, codifiche/decodifiche e sincronizzazione.

Talvolta alcuni strati sono vuoti e le funzionalità che dovrebbero offrire vengono implementate complessivamente in un unico strato superiore o inferiore.

EIB **(European Installation *Bus*)**

EIB[7] è stato sviluppato in Germania nel 1987 ed è il risultato della collaborazione tra *Insta*, *Merten* e *Siemens*.

Il suo obiettivo è la gestione di impianti industriali di grosse dimensioni ed è per questo motivo più adatto alla automazione di grandi edifici (*building automation*) piuttosto che a quella della casa (*home automation*).

Prevede una fase iniziale di configurazione dell'intero sistema che deve essere svolta da un tecnico specializzato attraverso strumenti appropriati.

Tutte le aziende che sviluppano dispositivi o sistemi EIB fanno parte di EIB*A (*EIB *Association)*.

I mezzi trasmissivi più utilizzati da questo sistema sono: *TP1 (Twisted Pair 1)*, *PL110 (Power Line 110KHz)*, *RF (Radio Frequency)* e *IR (Infrared Rays)*.

[7]http://www.eiba.org

***BatiBus* (*Batiment BUS*)**

BatiBus[8] è nato in Francia con la collaborazione di *Schneider*. Questo sistema *bus* può utilizzare solo *TP0 (Twisted Pair 0)* come unico mezzo trasmissivo.

***EHS* (European Home System)**

EHS[9] è il frutto di *Esprit*, un progetto promosso dall'Unione Europea al quale hanno partecipato numerose aziende quali *Philips*, *Siemens* e *Zanussi/Electrolux*.

Rispetto a EIB, abbandona il vincolo di configurazione manuale del sistema, sposando il concetto più moderno e dinamico di plug and play; questo sistema *bus* può usare quasi tutti i mezzi trasmissivi attualmente disponibili (*TP0*, *PL132*, *IR*, *CX*, etc.).

Dal processo di convergenza e di fusione di questi tre standard domotici europei che utilizzano un sistema di tipo *bus*, è nato e si è sviluppato *Konnex*.

2.1.2 *Konnex*

Il primo annuncio ufficiale dell'avvio di un processo di convergenza tra EIB, *BatiBus* e *EHS*, avvenne a *Birmingham* durante la fiera *Living Home del 19 marzo 1996*. Fu però durante il 1997 che fu stipulato l'accordo per la realizzazione di un unico

[8]http://www.bati*bus*.com
[9]http://www.ehsa.com

sistema *bus*, che costituisse l'unico standard Europeo in ambito di automazione degli edifici.

Lo scopo della convergenza era quello di incrementare il mercato della domotica, definendo un protocollo comune per l'automazione e assicurando l'interoperabilità dei tre sistemi.

Nel febbraio del 1999, otto aziende (tra cui *Siemens*, *Bosch*, *Merten*, *Hager* e *Schneider*) rilasciarono le specifiche di *Konnex* come singola forza europea per la standardizzazione delle reti domotiche.

Architettura

Il sistema *Konnex* fornisce le basi per lo sviluppo di applicazioni che si occupano di gestire l'ambiente domestico e, più in generale, quello di qualsiasi edificio.

Una premessa fondamentale per raggiungere questo obiettivo è quella di rappresentare l'ambiente come un sistema in cui ogni dispositivo costituisce una vera e propria applicazione distribuita: per questo motivo si parla di rete *Konnex*.

Come avviene per gli standard da cui deriva, anche le specifiche della rete *Konnex* seguono il modello architetturale standard *ISO/OSI* per definire gli strati, i protocolli e le funzionalità offerte ad ogni strato. In particolare le specifiche definiscono 5 dei complessivi 7 strati del modello *ISO/OSI*.

Lo standard *Konnex* specifica molti meccanismi per rende-

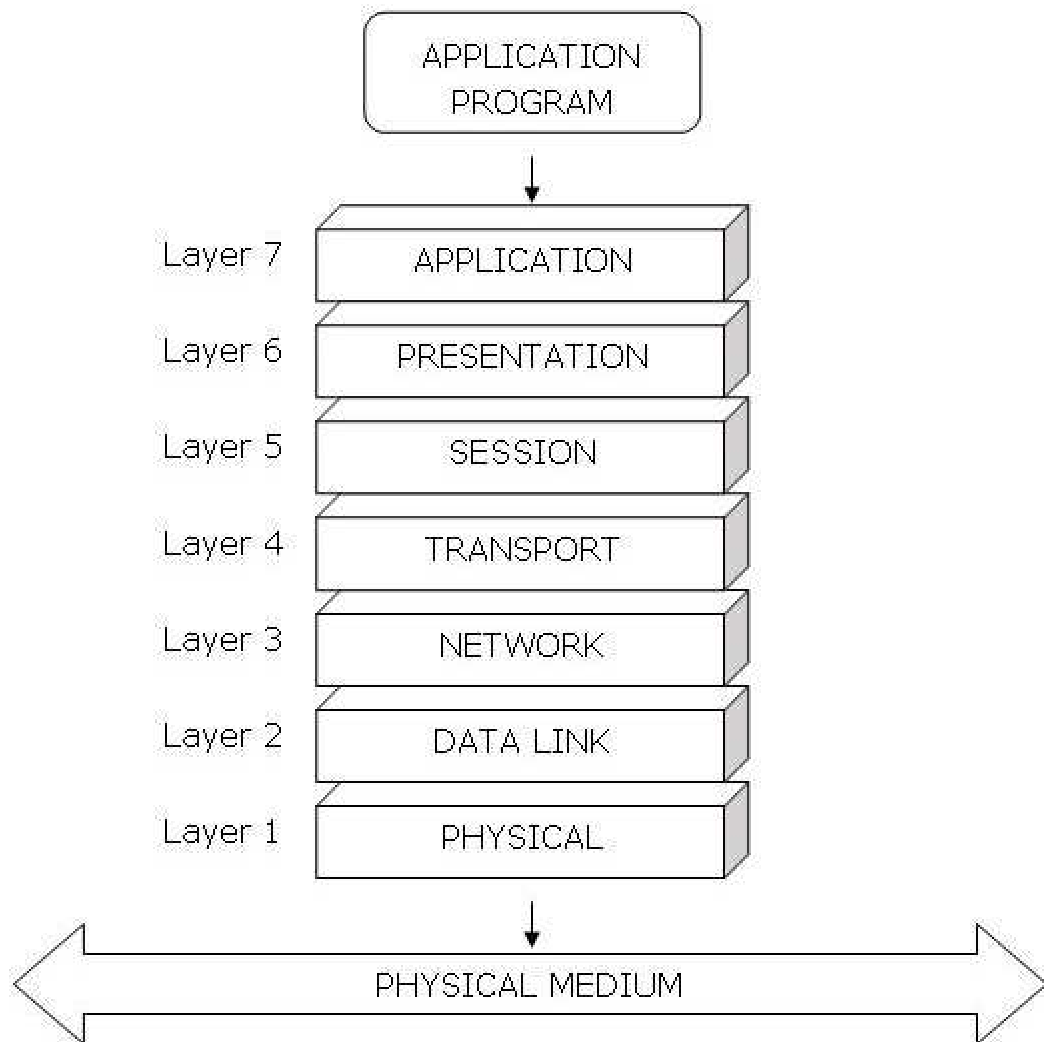

Figura 2.3: Modello architetturale del sistema Konnex.

re operativa la rete, consentendo così ai produttori una certa flessibilità nello sviluppo dei propri dispositivi.

Di seguito (figura 2.4) è riportata una panoramica del sistema *Konnex* che evidenzia sia le caratteristiche obbligatorie per la conformità allo standard, sia le scelte aperte lasciate al produttore. Lo schema non descrive i protocolli e le scelte tecniche, ma piuttosto individua quegli aspetti necessari per lo sviluppo di dispositivi e componenti.

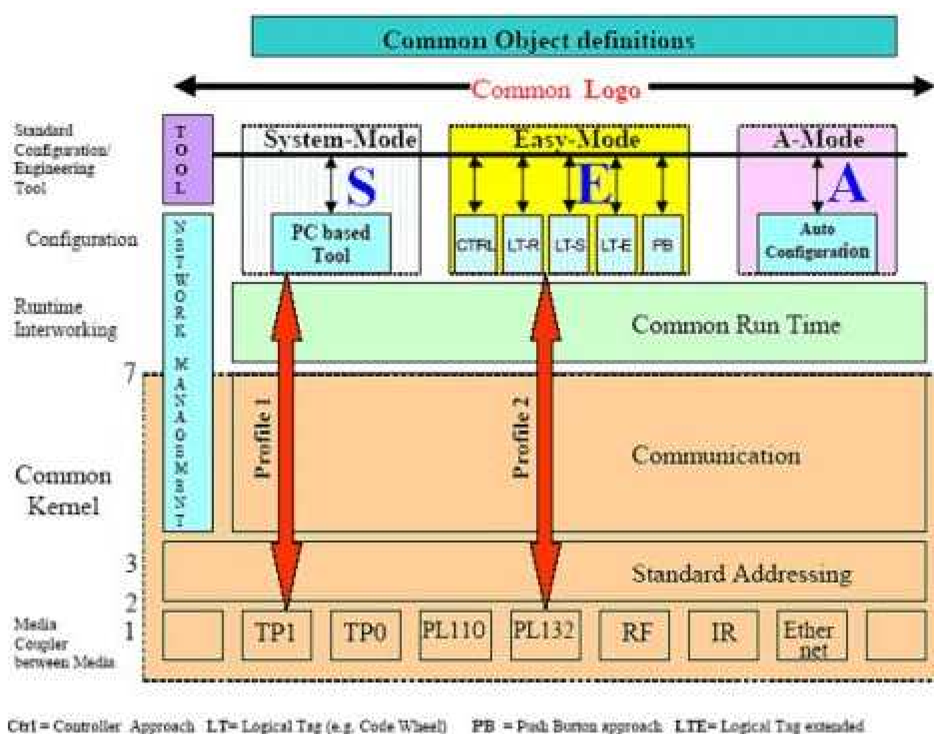

Figura 2.4: Il sistema konnex.

Descriviamo ora i principali moduli che compongono il modello:

- *communication system*:

deve supportare tutti i requisiti per una corretta comunicazione di rete in base al tipo di configurazione scelto per una particolare installazione del sistema;

- *interworking and distributed application models*:
  costituisce il *framework* essenziale per svolgere le attività necessarie all'automazione dell'edificio;

- *configuration and management*:
  gestisce accuratamente tutte le risorse presenti all'interno della rete e permette un collegamento logico tra applicazioni distribuite in esecuzione su nodi della rete (dispositivi) distinti;

- *concrete device models*:
  rappresenta l'insieme dei modelli di dispositivi raccolti in cosiddetti *profiles* (profili), che devono essere seguiti per la produzione di dispositivi conformi allo standard.

Alla base del concetto di applicazione proposto da *Konnex* c'è l'idea dei *datapoint*. Con questo termine vengono indicate tutte le variabili di processazione e controllo dell'intero sistema.

I *datapoint* possono costituire input, output, parametri, dati di tipo diagnostico, etc. e sono "contenuti" in *Group Objects* e in *Interface Object Properties*.

Il sistema di comunicazione del modello *Konnex* deve fornire un insieme di istruzioni ridotto ma completo per la gestione dei

datapoint; tipicamente queste istruzioni consentono la lettura e la scrittura dei valori contenuti nei *datapoint*.

Per garantire l'interfunzionalità all'interno del sistema, i *datapoint* devono implementare dei tipi standardizzati di dato, chiamati *datapoint type*, raggruppati in cosiddetti *functional block* (blocchi funzionali). Spesso i *functional block* e i *datapoint type* dipendono dal dominio applicativo, ma alcuni di essi possono essere di utilità generale (si pensi, ad esempio, alla codifica della data e del tempo all'interno di un timer).

L'accesso ai *datapoint* può seguire uno schema di tipo *unicast* o *multicast*, mentre il collegamento logico tra i *datapoint* delle applicazioni distribuite sulla rete, avviene tramite tre cosiddetti modelli di binding: *free*, *structured* e *tagged*. Descriviamo ora come questi modelli di collegamento possono essere combinati con i vari meccanismi di indirizzamento.

Durante il processo di installazione del sistema *Konnex* ci sono due fasi che richiedono un'attività di configurazione. La prima fase riuarda la configurazione della topologia della rete e dei singoli dispositivi che ne costituiscono i nodi; nella seconda invece si vanno a configurare le applicazioni distribuite sulla rete, comprendendo il collegamento tra i *datapoint* e il settaggio dei relativi parametri. Complessivamente, una particolare installazione del sistema *Konnex* ha i seguenti obiettivi:

- evidenziare un preciso schema di configurazione e collegamento;

- mappare questo schema nella scelta di un adeguato schema di indirizzamento;
- scegliere un insieme di procedure specifiche per la gestione delle risorse del sistema.

Alcune modalità di configurazione richiedono un processo di gestione della rete che coinvolge una comunicazione attiva sul *bus* (sia di tipo peer-to-peer che di tipo master-slave), mentre altre si limitano ad un'attività locale al dispositivo stesso.

Affinché sia praticabile un tipo di configurazione attiva, il sistema *Konnex* è equipaggiato con un potente insieme di strumenti per la gestione delle risorse della rete. Con il termine risorse si intendono sia dati relativamente semplici come indirizzi e parametri per la comunicazione, sia dati più complessi come tabelle di binding o veri e propri programmi applicativi.

Le attività correlate alla gestione della rete costituiscono un insieme di meccanismi per la scoperta, il settaggio e la ricezione, attraverso la comunicazione con il *bus*, dei dati necessari alla configurazione del sistema. Vengono impiegate le cosiddette *procedures* (procedure), ossia sequenze di messaggi, che permettono di accedere ad alcune risorse della rete esposte dai dispositivi e garantire così il corretto funzionamento di tutti i componenti della rete *Konnex*. Il modulo di gestione della rete sfrutta i servizi offerti dallo strato Application dello schema *ISO/OSI*; ogni dispositivo che implementa una particolare modalità di configu-

razione deve anche implementare i servizi e le risorse specifiche per un certo profilo.

Lo standard *Konnex* offre un ampio spettro di scelta ai produttori di componenti per quel che riguarda il supporto dei mezzi trasmissivi.

I mezzi trasmissivi supportati sono:

- *TP0* e *TP1*:
  il primo è ereditato dallo standard *BatiBUS*, mentre l'altro è il principale mezzo trasmissivo usato dallo standard EIB. Le caratteristiche principali sono che i dati e l'alimentazione vengono trasmessi su una coppia di canali, il trasferimento dei dati segue uno schema asincrono orientato al carattere e la comunicazione è bidirezionale half-duplex. Il rate di trasmissione varia dai 2400 bit/sec del TP0 ai 9600 bit/sec del *TP1.* Sia *TP0* che *TP1* implementano il protocollo *CSMA/CA* (*Carrier Sense Multiple Access/Collision Avoidance*) per l'accesso al mezzo fisico;

- *PL110* e *PL132*:
  il primo deriva da EIB, mentre l'altro è ereditato dallo standard *EHS.* Le principali caratteristiche consistono nella trasmissione asincrona dei dati e nel tipo di comunicazione bidirezionale *half-duplex.* I due mezzi differiscono in termini di frequenza, processo di decodifica dei dati e rate di trasmissione. Entrambi implementano il protocollo CSMA per gestire l'accesso al mezzo fisico.

- *RF*:
  è stato introdotto completamente dalle specifiche *Konnex* e consente la comunicazione wireless nella banda di frequenza radio a 868 MHz. Anche in questo caso la trasmissione è asincrona e la comunicazione può essere sia bidirezionale che unidirezionale half-duplex. Il protocollo utilizzato per l'accesso al mezzo è CSMA.

- *IR*:
  è stato ripreso da *EIB* e probabilmente costituirà le basi per gli sviluppi futuri del sistema *Konnex*. Oltre a questi mezzi trasmissivi, *Konnex* offre delle soluzioni standard ed integrate per gestire il protocollo *IP*, quali *Ethernet* (*IEEE 802.3*), *Bluethoot*, *Wi-Fi/Wireless LAN* (*IEEE 802.11*), *FireWire* (*IEEE 1394*).

Sulla base dello strato fisico e Data-link, tutti i componenti della rete *Konnex* condividono un cosiddetto Common Kernel (nucleo comune) che si basa sul modello *ISO/OSI* a 7 strati:

- *data-link general*:
  posto al di sopra dello strato Data-Link per medium, fornisce i meccanismi per il controllo dell'accesso al mezzo e del canale logico;

- *network layer*:
  offre un meccanismo di acknowledgement dei frames e ne controlla il numero di hop count (questo strato risulta par-

ticolarmente interessante per quei nodi che, all'interno della rete, hanno funzionalità di routing);

- *transport layer*:
  specifica 4 modelli di comunicazioni tra le entità della rete: uno-a-molti senza connessione (multicast), uno-a-tutti senza connessione (broadcast), uno-a-uno senza connessione e uno-a-uno orientato alla connessione;

- *application layer*:
  fornisce un vasto insieme di servizi, dedicati ai processi applicativi, che variano a seconda del tipo di comunicazione usato allo strato del trasporto.

Come evidente, gli strati *session layer* e *presentation layer* in questo caso sono vuoti. I servizi relativi ai modelli di comunicazione *punto-a-punto* e *broadcast* servono principalmente per la gestione della rete, mentre quelli relativi a comunicazioni multicast supportano le operazioni di runtime.

Come già visto, la gestione della rete consiste in un insieme di procedure per manipolare le risorse, mentre il *common kernel* fornisce un *framework* di servizi per adempiere tale proposito.

A causa del ruolo centrale che assumono all'interno del sistema, le risorse di rete meritano una descrizione più approfondita. Si possono distinguere risorse per il controllo delle applicazioni e risorse di sistema (o di configurazione), che comprendono inidirizzi e parametri informativi con lo scopo di agevolare le attività

di comunicazione. I cosiddetti Interface Objects forniscono un importante *framework*, indipendente dall'implementazione, per definire le risorse del sistema, i cui elementi individuali possono essere modellati come *datapoint*.

Da un punto di vista generico, l'installazione del sistema *Konnex* consiste semplicemente in un insieme di dispositivi connessi tra loro attraverso il *bus*. Tutte le funzionalità finora descritte sono realizzate in e per mezzo di questi dispositivi. A questo proposito si possono distinguere vari modelli logico-architetturali che un nodo della rete deve rispettare. Questi modelli variano in accordo con le principali caratteristiche del nodo, quali la gestione, la modalità di configurazione e il suo ruolo all'interno della rete; per quanto riguarda quest'ultimo aspetto si possono distinguere *application device*, *configuration master*, *router*, *gateway*, etc. Il sistema *Konnex* standardizza anche alcuni modelli di dispositivi che rivestono ruoli più generali all'interno della rete (general-purpose device), come *Bus Coupling Unit* (*BCU*) o *Bus Interface Module* (*BIM*), impiegati principalmente come interfaccia verso il *bus*. I modelli di dispositivi, insieme alle caratteristiche delle modalità di configurazione, sono complessivamente rappresentati all'interno dei *profiles*.

Vi possono essere diversi meccanismi di identificazione per accedere ai dispositivi presenti all'interno della rete: attraverso un indirizzo individuale o un numero di serie univoco proprio del dispositivo, in base alla modalità di configurazione; attraverso

altri tipi di informazioni direttamente ricavabili dal dispositivo, come l'identificativo del prodotto (dipendente dal produttore) o l'identificativo delle funzionalità offerte (indipendente dal produttore).

L'unicità dei numeri seriali è garantita da *Konnex* Association Certification Department.

Modelli architetturali

Communication System
Konnex è a tutti gli effetti una rete distribuita in cui trovano spazio un massimo di 65536 dispositivi e *datapoint* indirizzabili all'interno del sistema; ogni dispositivo è codificato da un indirizzo individuale di 16 bit. La topologia, o la struttura della sottorete, consente di disporre 256 dispositivi su una line (linea); più lines possono essere raggruppate tra loro attraverso una *main line* (linea principale) all'interno di un'area. Un dominio completo è costituito da 15 areas collegate tra loro attraverso una *backbone line*. La struttura topologica della rete si riflette nella struttura numerica degli indirizzi individuali che, tranne qualche eccezione, identifica in modo univoco un nodo all'interno del sistema. La figura di seguito (figura 2.5) mostra questa relazione tra la topologia e la struttura degli indirizzi.

Per gestire la rete e le risorse esposte dai dispositivi, *Konnex* utilizza una combinazione di servizi che si basano su comunicazione *broadcast* e *unicast punto-a-punto*. Molto spesso, du-

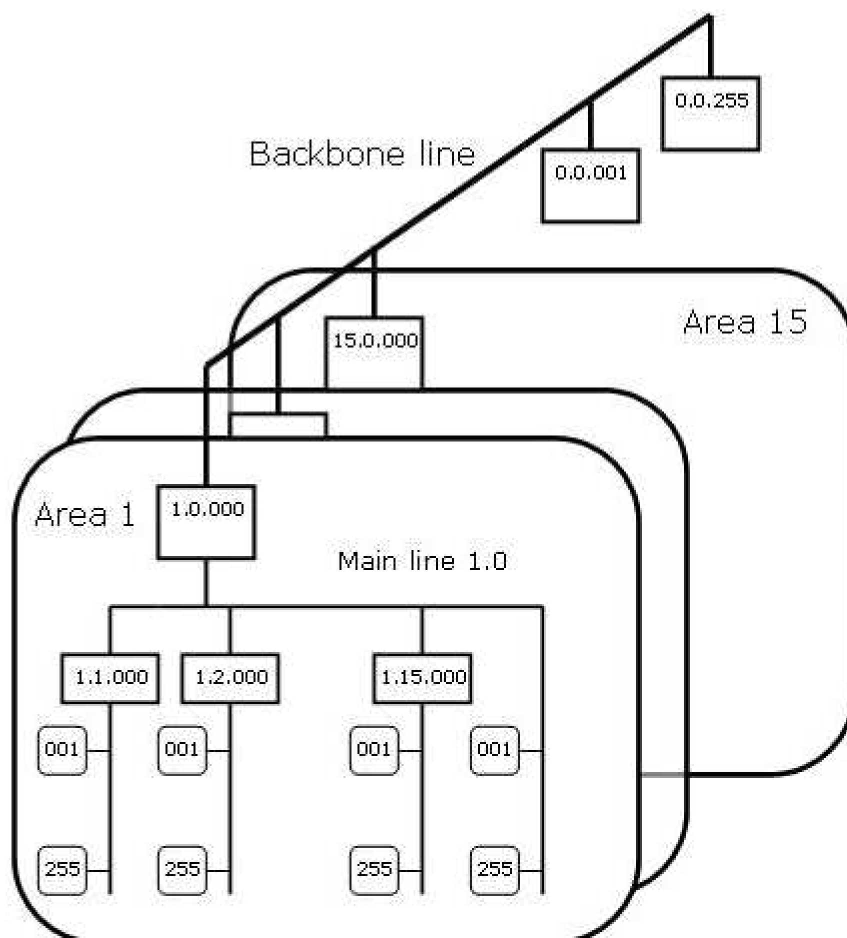

Figura 2.5: Topologia e schema di indirizzamento della rete Konnex.

rante la fase di installazione, ad ogni dispositivo viene assegnato un indirizzo individuale unico via broadcast, che successivamente viene utilizzato per identificarlo in tutte le comunicazioni *punto-a-punto*.

Konnex supporta completamente un modello di indirizzamento multicast, attraverso cosiddetti *group addresses* (indirizzi di gruppo); tale modello rappresenta il principale schema di comunicazione durante la fase esecutiva del sistema. Il suppor-

to completo al multicast si traduce nel fatto che *Konnex* non limita il raggruppamento dei dispositivi: ogni dispositivo può rendere pubblici molti *datapoint* distinti, detti *group communication Objects*, che poi possono essere raggruppati in maniera indipendente in cosiddette variabili condivise; su una variabile condivisa, poi, possono essere invocate sia operazioni di lettura che di scrittura. *Konnex* propone uno spazio di indirizzamento codificato in 16 bit per identificare le variabili condivise; ciò significa che un'installazione può supportare fino a 65536 variabili condivise (o *group addresses*) senza limitarne il numero di istanze locali.

Il messaggio *Konnex* è codificato in frame che rappresentano l'unità di informazione trasferita sul *bus*. A seconda del mezzo fisico, e quindi di particolari tecniche per la modulazione della frequenza o per il controllo dell'accesso al mezzo, il telegramma può presentare al suo inizio un cosiddetto preambolo.

Descriviamo ora le funzioni dei singoli campi che costituiscono il telegramma:

- *Control Field*:
  determina la priorità del telegramma e distingue i frame standard da quelli estesi; *Source Address*: rappresenta l'indirizzo inviduale del mittente;

- *Destination Address*:
  rappresenta l'indirizzo del/i destinatario/i: può essere in-

dividuale (per le comunicazioni *unicast*) o di gruppo (per le comunicazioni *multicast*);

- *Address Type*:
  specifica il tipo di indirizzo destinazione (*unicast* o *multicast*);

- *Hop Count*:
  consiste in un contatore che viene decrementato ogni volta che il frame transita attraverso un router (quando il contatore raggiunge lo 0, l'intero frame viene scartato per evitare che cicli indefinitamente all'interno della rete);

- *TPCI* (*Transport layer Protocol Control Information*):
  controlla le comunicazioni stabilite allo strato del trasporto (ad es. costruire e mantenere una connessione *punto-a-punto*);

- *APCI* (*Application layer Protocol Control Information*):
  identifica l'insieme dei servizi dello strato applicativo (*Read*, *Write*, *Response*, etc.) disponibili in accordo allo schema di indirizzamento e di comunicazione;

- *data*:
  contiene i dati trasportati dal telegramma che, in base al valore *APCI* e allo schema di indirizzamento, possono raggiungere un massimo di 14 ottetti;

- *Framecheck*:

assicura l'integrità e la consistenza del telegramma da trasmettere.

Distributed Application Model

Konnex modella un sistema applicativo distribuito (*Distributed Application Model*) dei dispositivi presenti all'interno della rete, attraverso l'invio e la ricezione dei *datapoint* che questi contengono. Il sistema risulta attivo quando i *datapoint* appartenenti a diversi dispositivi vengono collegati (operazione di binding) tramite un identificatore comune, come accade, ad esempio, nel caso dei *group addresses multicast*; in questo modo le applicazioni locali ad ogni dispositivo possono scambiarsi dati ed informazioni. Quando l'applicazione locale ad un certo dispositivo vuole aggiornare il valore di un proprio *datapoint* esportabile, invia un messaggio di tipo write con l'indirizzo destinazione del *datapoint* ed il nuovo valore. I *datapoint* corrispondenti a questo indirizzo, e quindi collegati con il *datapoint* mittente, riceveranno il nuovo valore ed informeranno le proprie applicazioni locali dell'avvenuto cambiamento. In questo modo, i processi locali ai dispositivi, collegati tra loro attraverso i *datapoint*, costituiscono una vera e propria applicazione distribuita sulla rete *Konnex*. Il sistema prevede 3 schemi per il collegamento dei *datapoint* in base al fatto che il valore dell'indirizzo trasporti o meno informazioni di tipo semantico; tali schemi sono denominati *free binding*, *structured binding* e *tagged binding*, e saranno descritti in dettaglio più avanti. Descriviamo ora le varie modalità per realizzare un

datapoint.

Il principale esempio di realizzazione di *datapoint* è fornito dai *group object*; come suggerisce il nome, a questi *datapoint* si accede attraverso un indirizzamento standard multicast che prevede l'uso dei *Group Address* come collegamento. I *Group Object* possono essere usati con tutti e tre gli schemi di collegamento (*free*, *structured* e *tagged binding*). Per rispettare alcuni requisiti aggiuntivi, *Konnex* fornisce un'altra forma di realizzazione di *datapoint*, che si traduce in una proprietà appartenente ad un *Interface Object*. Un Interface Object rappresenta il contenitore di un insieme di *datapoint* che condividono alcune proprietà definite in un'interfaccia comune. Mentre i *Group Object* di un nodo costituiscono un insieme piatto di *datapoint* indirizzabili direttamente, ogni proprietà di un Interface Object è relativa ad un particolare oggetto del nodo e può essere rappresentata secondo lo schema `<ObjectReference>.<Property>`.

Per poter stabilire un collegamento tra i partner di una comunicazione, sono necessari due fasi: la selezione di un indirizzo e l'assegnazione di tale indirizzo ai *datapoint* che devono essere collegati. Per quanto riguarda la prima fase, sia lo schema *free binding* che quello *structured binding* assumono che vi sia un meccanismo di indirizzamento libero; ciò significa che il valore numerico scelto come indirizzo non porta con sé informazioni riguardo alla semantica dell'applicazione. Si assume inoltre che a tutti i *datapoint* che desiderano comunicare tra loro venga

assegnato il medesimo indirizzo. La seconda fase, ovvero l'assegnazione dell'indirizzo selezionato, varia a seconda dello schema di collegamento. Nel caso di schema *free binding*, non esistono delle regole a priori che stabiliscano quali *datapoint* possano essere collegati l'un l'altro, tranne per alcuni vincoli che garantiscono la consistenza dei *datapoint Types.* Questo tipo di schema di collegamento è al centro della modalità di configurazione *S-Mode.* Nel caso invece di schema structured binding, il modello applicativo delle specifiche *Konnex* stipula un preciso schema per il collegamento di un insieme di *datapoint* che solitamente coincide con un *functional block* o un *channel.* La modalità di configurazione *A-Mode* si basa su questo schema.

A differenza dei due precedenti modelli di collegamento, in questo schema la selezione dell'indirizzo contiene informazioni di tipo semantico; il *semantic datapoint identifier*, una parte del valore numerico che lo rappresenta, identifica i *datapoint* partner della comunicazione. La modalità di configurazione *E-Mode* adotta questo schema di collegamento, il quale è strettamente legato al concetto di logical tag o zoning: il logical tag rappresenta una porzione dell'indirizzo che specifica i partners di una comunicazione a livello del dispositivo. Quindi, per un certo *datapoint*, il *semantic datapoint identifier* viene fissato dal modello applicativo, mentre il logical tag viene assegnato in locale al dispositivo durante la procedura di installazione. Quando più *datapoint* condividono lo stesso logical tag formano un gruppo e

possono sfruttare tutte le caratteristiche di una comunicazione multicast. Il sistema *Konnex* prevede 3 possibilità per realizzare il modello tagged binding:

- il modello viene mappato nello schema di indirizzamento di gruppo secondo la seguente conversione:
  `<Group_Address> = <Zone>.<SemanticID>;`
- il modello viene mappato sullo schema di indirizzamento individuale attraverso Interface Objects:
  `<Target>.<SemanticID>` dove
  `<Target> = <Individual_Address >` e
  `<SemanticID> = <ObjectReference>.<PropertyID>;`
- il modello viene mappato sullo schema di indirizzamento esteso attraverso Interface Objects.

Interworking Model
Il modello di funzionamento (*interworking model*), rappresenta una delle caratteristiche più importanti del sistema *Konnex*; grazie a questo è possibile garantire la massima integrazione tra dispositivi (e quindi applicazioni) appartenenti a domini eterogenei. Il modello di funzionamento descrive il comportamento delle applicazioni dal punto di vista della rete; in altre parole specifica la sua interfaccia di accesso tramite *datapoint*. Se all'interfaccia dell'applicazione si aggiunge il suo comportamento in termini di stato interno, input e output, si costruisce il cosiddetto *functional block*, che rappresenta una descrizione di ogni

aspetto locale dell'applicazione distribuita. Le specifiche di un *functional block* prevedono che al suo interno siano assegnati ad ogni *datapoint* una descrizione esplicativa ed il tipo richiesto (*datapoint type*), che stabilisce il formato dei dati che il *datapoint* invia o riceve dal *bus*. I principali tipi di dato standardizzati dal sistema *Konnex* sono:

- *binary value* (*boolean*),
- *relative control* (%),
- *analog value* (*long* e *float*),
- *counter value* (*signed* e *unsigned integer*),
- *date and time*,
- *status* (*bit*).

Configuration and Management

Le specifiche *Konnex* offrono diverse modalità di configurazione del sistema; ognuna di queste è sviluppata con l'obiettivo di lasciare un alto grado di libertà ai produttori di dispositivi e, allo stesso tempo, garantire l'interoperabilità del sistema. Descriviamo di seguito ciascuna di queste modalità, in termini di caratteristiche tecniche e strumenti per realizzarle.

1. *System-Mode*: (*S-Mode*)
   i dispositivi implementati secondo questo modello offrono il più versatile processo di configurazione, perché lasciano

il maggior margine di azione all'installatore del sistema. Di contro, è sempre necessaria la presenza di un tecnico qualificato e certificato *Konnex* per poter effettuare un'installazione adeguata. La procedura di collegamento e di configurazione dell'applicazione è lasciata ad un potente configuration master; solitamente questo ruolo è svolto da un insieme di strumenti software per PC (ETS = *EIB* Tools Software), distribuiti da *Konnex* Association. Con l'ausilio di questi strumenti, e in particolare di ETS per la gestione dei progetti, l'installatore può configurare i dispositivi e rendere operativo il sistema. Per ottenere tutte le informazioni relative ai dispositivi, il software ETS fa uso di un database che rappresenta tutte le possibili funzionalità dei dispositivi supportati; queste informazioni vengono create e mantenute da ogni produttore di dispositivi, utilizzando un altro strumento software della famiglia ETS. Di solito i produttori forniscono i propri database ai clienti: in questo modo gli installatori possono importare nel database ETS le informazioni provenienti da diversi produttori e costruire così un sistema che faccia uso di dispositivi di marchi distinti.

Gli strumenti della famiglia *ETS* per lo sviluppo e la progettazione della rete *Konnex* offrono le seguenti funzionalità:

- *binding*:

  per assegnare gli indirizzi di gruppo che abilitino la co-

municazione *Group Object* tra *functional block* (i *group object* sono messi in relazione se condividono il medesimo *datapoint type*);

- *parameterisation*:
  per settare i parametri del dispositivo in accordo con le specifiche fornite dal produttore;

- *application program download*:
  per caricare il codice eseguibile del programma applicativo all'interno della *EEPROM* del dispositivo attraverso un'interfaccia tra il PC ed il *bus* (*RS232*, *USB*).

2. *Easy-Mode* (*E-Mode*)
   Anche per la modalità *E-Mode* è richiesta la presenza di un tecnico specializzato al momento dell'installazione. All'interno della modalità *E-Mode* si possono distinguere cinque ulteriori modalità, che andiamo ora a descrivere nello specifico.

   - *CONTROLLER Mode* (*CTRL-Mode*):
     questa modalità di configurazione è indicata per l'installazione di un numero limitato di dispositivi su un unico segmento logico del mezzo fisico utilizzato. Un'installazione che fa uso di questo modello deve prevedere l'impiego di un particolare dispositivo, detto controller, che ha il compito di gestire il processo di configurazione. Durante la configurazione, il ruolo del controller è quel-

lo di stabilire i collegamenti tra i cosiddetti *Channels* (canali), che rappresentano l'insieme dei *group object* necessari per adempiere una certa funzione.

Il controller assegna gli indirizzi individuali ai dispositivi, calcola i collegamenti a livello dei *datapoint* attraverso regole standardizzate dalle specifiche, ed infine setta i parametri per i dispositivi presi in considerazione.

- *Push Button Mode* (*PB-Mode*):
  anche questo modello, come il precedente, è indicato per installazioni che prevedono un numero limitato di dispositivi. Tuttavia, in questo caso, non c'è la necessità di eleggere un particolare dispositivo che gestisca il processo di configurazione: ogni dispositivo che implementa questo schema include infatti un meccanismo di configurazione specifico per la propria applicazione. Durante la configurazione, l'installatore seleziona i dispositivi le cui funzioni devono essere collegate in maniera dipendente dal produttore del dispositivo stesso.

- *Logical Tag Reflex Mode* (*LTR-Mode*):
  utilizzando questa modalità non è necessario alcuno strumento di configurazione: le funzioni situate in dispositivi diversi sono collegate tra loro attraverso l'assegnamento del medesimo tag.

- *Logical Tag Supervised Mode* (*LTS-Mode*):
  come per *LTR*, anche in questo modello non è necessario alcuno strumento di configurazione, ma è obbligatorio eleggere un supervisore per ogni applicazione, che comunque può controllarne più di una. Il supervisore utilizza dei meccanismi standard per assegnare i corretti *group addresses* di cui un dispositivo necessita a runtime. Le funzioni presenti in dispositivi distinti che hanno lo stesso *group address* sono collegate e possono interoperare.

- *Logical Tag Extended Mode* (*LTE-Mode*):
  questo modello è attualmente limitato ad una sola categoria di applicazioni (*HVAC*), che necessita di un vasto insieme di strutture dati; questi dati vengono scambiati attraverso Interface Objects usando il telegramma esteso, previsto dal protocollo *Konnex*.

3. *Automatic-Mode* (*A-Mode*):
   al contrario dei precedenti modelli di configurazione, *A-Mode* non richiede la presenza di un installatore: tale modalità include infatti meccanismi che consentono ai dispositivi di stabilire automaticamente i collegamenti necessari. Questa caratteristica è stata elaborata al fine di modellare una sorta di *plug and play* che risolva le conseguenze dovute alla mobilità dei dispositivi. Gli applicativi sono in grado di acquisire indirizzi individuali in modo autonomo

e prevedono la presenza di un controller dedicato (*Application Controller*), che solitamente svolge anche il ruolo di *configuration master*. La configurazione dei *group Address* è eseguita dinamicamente dal *configuration master*, il quale richiede le informazioni necessarie attraverso *Interface Object* in una comunicazione *punto-a-punto*.

Concrete Device Models

Le specifiche del sistema *Konnex* non sono altro che un *framework* dal quale estrarre le informazioni per rendere un certo dispositivo conforme allo standard, indipendentemente dal dominio applicativo. Per mantenere il sistema coerente con le specifiche e per consentire la certificazione di prodotti standard, queste informazioni sono state raggruppate in cosiddetti *profiles* (profili); i profili sono limitati, e sono stati ritagliati attorno alle caratteristiche e alle necessità dei membri della comunità *Konnex*. I profili attualmente disponibili sono: dispositivi *S-Mode*, dispositivi *E-Mode*, dispositivi *A-Mode* e client per la gestione di qualsiasi profilo di dispositivi. Per certificare un prodotto, è necessario che il fabbricante dichiari a quale profilo tale dispositivo è ritenuto conforme, dopodiché il processo di certificazione *Konnex* eseguirà gli opportuni test per verificarne l'effettiva conformità.

Strumenti software

Eib Tools Software (ETS)

Il più importante strumento messo a disposizione dalla famiglia ETS[10] è quello per la progettazione e la configurazione delle installazioni *Konnex*: ETS3. Questo strumento consente di svolgere essenzialmente due attività: la prima consiste nella progettazione, configurazione e gestione di installazioni *S-Mode*; la seconda attività riguarda invece l'integrazione tra reti *Konnex* con diversa modalità di configurazione. Il generico progetto ETS costituisce una rappresentazione *off-line* dell'installazione del sistema; più in dettaglio, il progetto è un database che contiene una ricca descrizione delle informazioni contenute nel sistema installato. Se il software ETS è connesso al *bus* (attraverso *RS232* o *USB*) può essere utilizzato da un installatore, tramite interfaccia grafica, come un potente *configuration master*. Inoltre, *ETS* è in grado di scansionare l'installazione, e di scoprire e integrare i dispositivi appartenenti a diversi profili. Ricordiamo infine che esiste una versione di *ETS* abilitata all'accesso Internet (*iETS*), che offre le stesse funzionalità della versione tradizionale attraverso un collegamento IP remoto.

e-tool environment Component (eteC)

Il software *ETS* è costruito su un *framework* basato su componenti *DCOM* (*Distributed Component Object Model*) per piattaforme *Windows*, denominato *e-tool environment Component*

[10] http://www.eiba.com/en/ets3/index.html

(*eteC*). Questo *framework* fornisce un'interfaccia per i programmi applicativi (API = *Application Programming Interface*) ed un modello ad oggetti, essenziali per l'accesso *on-line* ed *off-line* alle risorse *Konnex*. Un primo beneficio che si trae dal mapping tra la logica del sistema *Konnex* e quella degli oggetti *DCOM* è che l'interfaccia eteC così ottenuta può essere usata con la maggior parte dei linguaggi di programmazione disponibili (C, C++, Java, Visual Basic, etc.). Anche i linguaggi di scripting, come JavaScript, sono supportati dal *framework eteC*. Le due principali componenti *eteC* disponibili per gli sviluppatori sono *Eagle* e *Falcon*. *Eagle* ha il compito di effettuare il mapping tra la struttura fisica del database *ETS* ed un modello di dati astratto basato sul concetto di Domain Objects. *Falcon*[11] invece è una libreria *DCOM* a 32 bit che consente agli applicativi che girano in ambiente Windows di comunicare attraverso la rete *Konnex/EIB*. Gestisce e implementa in modo trasparente all'applicativo tutti i meccanismi specifici del protocollo di comunicazione; l'applicativo risulterà così notevolmente più semplice da scrivere e potrà comunicare con i dispositivi *Konnex/EIB* facendo delle semplici chiamate di funzione (*API DCOM*). *Falcon* implementa tutte le funzioni previste dallo standard *Konnex/EIB*, e consente quindi di gestire in modo completo una rete *EIB*, per quanto riguarda sia la configurazione che la supervisione ed il controllo.

[11]http://www.eiba.com/en/software/falcon/index.html

Per quanto concerne la configurazione, la nuova versione del software applicativo *ETS* (*ETS3*) utilizza *Falcon* per l'accesso alla rete *EIB*. Ad esempio, per determinate applicazioni caratterizzate sempre dalla stessa tipologia e numero di dispositivi, è possibile utilizzare *Falcon* per realizzare un software applicativo per la configurazione automatica della rete *EIB*, eliminando così la fase di configurazione manuale con *ETS*. Nelle attività di supervisione e controllo, *Falcon* consente inoltre di accedere in lettura e scrittura a tutti i dati resi disponibili dai dispositivi *Konnex/EIB*.

2.1.3 *UPnP 1.0* (*Universal plug and play*)

Architettura

Il middleware *UPnP* costituisce uno dei più importanti esempi di tecnologie domotiche basate sullo *stack* dei protocolli *TCP/IP*. A differenza dei sistemi *bus* analizzati finora, *UPnP* non necessita di componenti hardware particolari, ma sfrutta i comuni meccanismi di accesso alla rete *IP* per costituire il proprio modello di comunicazione. La tecnologia *UPnP* definisce un'architettura per lo sviluppo di reti peer-to-peer tra dispositivi wireless, apparecchiature domestiche ed elaboratori generici. I protocolli utilizzati da *UPnP* sono quelli standard di Internet come *IP*, *TCP*, *UDP*, *HTTP*, *XML* e *SOAP*; questo consente il supporto di qualsiasi mezzo fisico per cui sia disponibile lo *stack TCP/IP*, e lo sviluppo di software attraverso molteplici linguaggi di program-

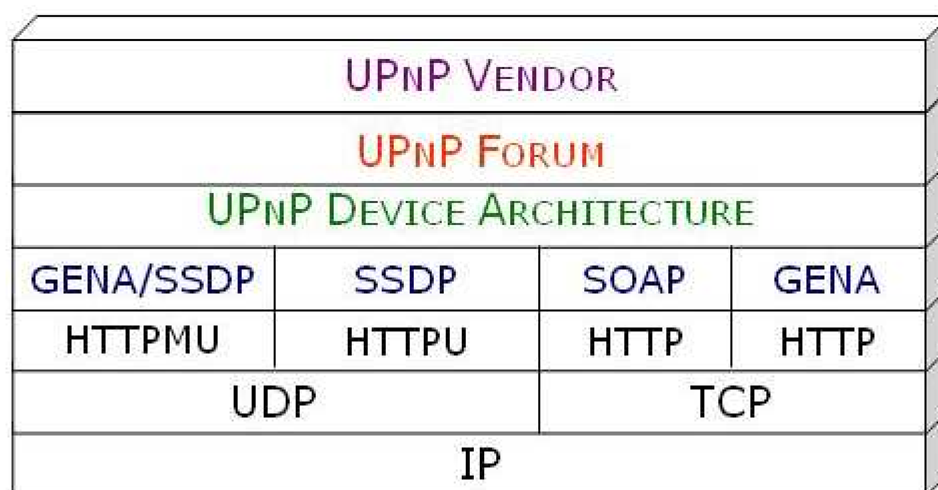

Figura 2.6: Struttura del frame Konnex *allo strato Data-Link.*

mazione, su qualunque ambiente operativo implementi *TCP/IP*. Alla base dell'organizzazione *UPnP* si trova la cosiddetta *UPnP Device Architecture*, ossia l'interfaccia comune verso i protocolli standard definiti dalle specifiche. Le attività proposte da *UPnP* sono regolamentate da *UPnP* Forum, un'iniziativa guidata da *Microsoft* e da oltre 700 partner industriali. Le scelte promosse dai singoli produttori vengono gestite da *UPnP Vendor*. In figura 2.6 viene mostrato lo *stack* dei protocolli utilizzati dallo standard *UPnP*.

UPnP identifica due classi di dispositivi: *controlled device* (o semplicemente device) e *control point*. Un *controlled device* svolge il tipico ruolo di un server, rispondendo alle richieste provenienti da uno o più *control point*, che assume quindi il ruolo di client. Entrambe le categorie di dispositivi possono essere implementate su varie piattaforme, come PC, PDA o sistemi

embedded.

L'interazione generale in uno scenario *UPnP* prevede 6 fasi distinte (*addressing*, *discovery*, *description*, *control*, *eventing* e *presentation*), che andiamo ora a analizzare nel dettaglio.
Fasi di interazione

1. *addressing*:
   alla base del funzionamento di una rete *UPnP* c'è l'indirizzamento IP. Ogni dispositivo deve essere provvisto di un client *DHCP* (*Dynamic Host Configuration Protocol*): quando un dispositivo viene collegato alla rete per la prima volta, deve ricercare un server *DHCP* che provveda ad assegnargli un indirizzo IP; se la rete non è configurata e non vi sono server *DHCP* disponibili, il dispositivo utilizzerà il meccanismo di AutoIP per ottenere un indirizzo valido;

2. *discovery*:
   quando il dispositivo si è collegato alla rete *UPnP* ed ha ottenuto un indirizzo IP valido, pubblica i servizi che espone verso tutti i control points presenti. Allo stesso modo, quando un control point viene aggiunto alla rete, potrà ricercare tutti i dispostitivi a cui è interessato. Entrambe le interazioni sono veicolate da *SSPD* (*Simple Service Discovery Protocol*), il protocollo di discovery sviluppato dallo standard;

3. *description*:

dopo che un control point ha scoperto la presenza di un dispositivo, può conoscerne le caratteristiche ottenendo la sua descrizione in formato *XML*, a partire dalla *URL* (*Uniform Resource Locator*) fornita dallo stesso dispositivo durante la fase di discovery. La struttura di un dispositivo è gerarchica: l'elemento principale è il cosiddetto root device che, al suo interno, può contenere cosiddetti embedded device (dispositivi logici); ogni embedded device racchiude alcune funzionalità che espone verso la rete sotto forma di services (servizi). La descrizione *XML* del dispositivo è suddivisa in due parti: la prima contiene informazioni relative al produttore (ad es. il marchio, il numero seriale, la URL della fabbrica, etc.); la seconda specifica i dettagli di ogni servizio esposto da quel particolare dispositivo (ad es. la lista dei comandi previsti per quel servizio, il numero ed il tipo di parametri ammessi, etc.);

4. *control*:

   quando il control point ha ottenuto tutte le informazioni necessarie per un certo dispositivo, può invocare le azioni corrispondenti ai servizi che questo espone. L'interazione avviene tramite lo scambio di messaggi SOAP (Simple Object Access Protocol), seguendo un modello RPC (Remote Procedure Call);

5. *eventing*:

   la descrizione di un dispositivo conforme allo standard *UPnP*

contiene, tra l'altro, una lista di variabili che ne modellano lo stato interno. A seguito di particolari e contingenti condizioni, queste variabili possono cambiare valore e il servizio a cui si riferiscono pubblica alla rete questi aggiornamenti. Un control point deve sottoscriversi ad un meccanismo di eventi per ricevere le notifiche dei cambiamenti delle variabili di stato appartenenti ad un certo dispositivo. I messaggi di evento sono anch'essi espressi secondo il formalismo standard *XML* e veicolati dal protocollo GENA (General Event Notification Architecture);

6. *presentation*:
   il control point può avere bisogno di ottenere informazioni aggiuntive su un particolare dispositivo; tali informazioni, se presenti, sono contenute in una pagina web indicata in un'opportuna *URL* nel file *XML* di descrizione del dispositivo.

Attualmente è in corso di rilascio *UPnP Device Architecture 2.0*, che si baserà totalmente sui protocolli standard definiti dal *framework* dei *web service.*

Capitolo 3

I Web Service

L'implementazione del *framework domoNet* avrebbe dovuto basarsi su un modello *SOA* (*Service Oriented Architecture*), in cui i servizi avrebbero coinciso con le funzionalità offerte dai dispositivi presenti nella casa. Alla luce di questo e del grado di interoperabilità raggiunto con lo sviluppo dei *web service* nei contesti applicativi *SOA*, si è scelto di porre i *web service*, e le tecnologie standard ad essi correlate, come fondamenta del *framework domoNet*.

La rivoluzione che l'introduzione dei *web service* sta generando nel modello di interazione application-application, sembra destinata ad avere le stesse proporzioni di quella che, nel decennio scorso, ha riguardato l'interazione user-application grazie all'impiego del *World Wide Web*. I *web service*, ed il paradigma di interazione su cui si basano, consentono notevoli riduzioni di costi per le attività di e-business, eliminando molte delle spese necessarie per garantire l'interoperabilità tra soluzioni applicative

diverse.

La chiave di questo potenziale successo risiede in un modello di interazione comune application-application, basato su collaudate tecnologie, come *HTTP* (*HyperText Transfer Protocol*), e standard più o meno recenti, quali *XML* (*eXtensible Markup Language*), *SOAP* (*Simple Object Access Protocol*), *WSDL* (*web service description language*) e UDDI (Universal Description Discovery and Integration).

Grazie a questo modello comune, i *web service* consentono di integrare più rapidamente ed efficacemente le applicazioni in un ambiente distribuito; tale integrazione avviene ad uno strato molto alto dello *stack* protocollare, e perciò si concentra più sul concetto di servizio che non su quello di protocollo di rete.

3.1 Definizione

Con il termine *web service* si definisce un'interfaccia che descrive un insieme di Services (servizi/operazioni) accessibili in un ambiente distribuito attraverso un protocollo basato su messaggi XML standardizzati. Un *web service* è rappresentato dalla corrispondente Service Description (descrizione dei servizi forniti) attraverso una grammatica XML standard; tale descrizione fornisce tutti i dettagli necessari per interagire con i servizi esposti dal *web service*, compreso i formati dei messaggi, i protocolli di trasporto e la locazione.

L'interfaccia nasconde i dettagli implementativi del servizio

e può così essere usata indipendentemente sia dalla piattaforma hardware/software, che dal linguaggio di programmazione con cui è implementata.

L'architettura dei *web service* è basata sull'interazione tra tre entità: *service provider*, *service registry*, *service requestor*. Le interazioni coinvolgono tre operazioni: *publishing* (pubblicazione), *finding* (ricerca) e *binding* (collegamento). Le tre entità e le tre operazioni interagiscono attraverso i due artefatti dei *web service*: il modulo software che implementa il *web service* (o semplicemente *Service*) e la corrispondente *service description*. In uno scenario tipico, un *service provider* ospita un modulo software, accessibile da remoto, che implementa il *web service* e definisce una *service description* per il *web service* di cui fornisce l'implementazione, pubblicando tale descrizione ad un *service requestor* o ad un *service registry* (operazione di *publishing*). Il *service requestor* ottiene la *service description* a cui è interessato attraverso una ricerca locale o diretta ad un *service registry* (operazione di *finding*); tramite la *service description* stabilisce un collegamento con il *service provider*, e può così interagire direttamente con l'implementazione del *web service* (operazione di *binding*). I ruoli di *service provider* e *service requestor* si distinguono da un punto di vista puramente logico-funzionale, e perciò possono risiedere contemporaneamente sulla medesima macchina. La figura di seguito (figura 3.1) illustra le operazioni, le entità che ne fanno uso e come queste interagiscono tra loro.

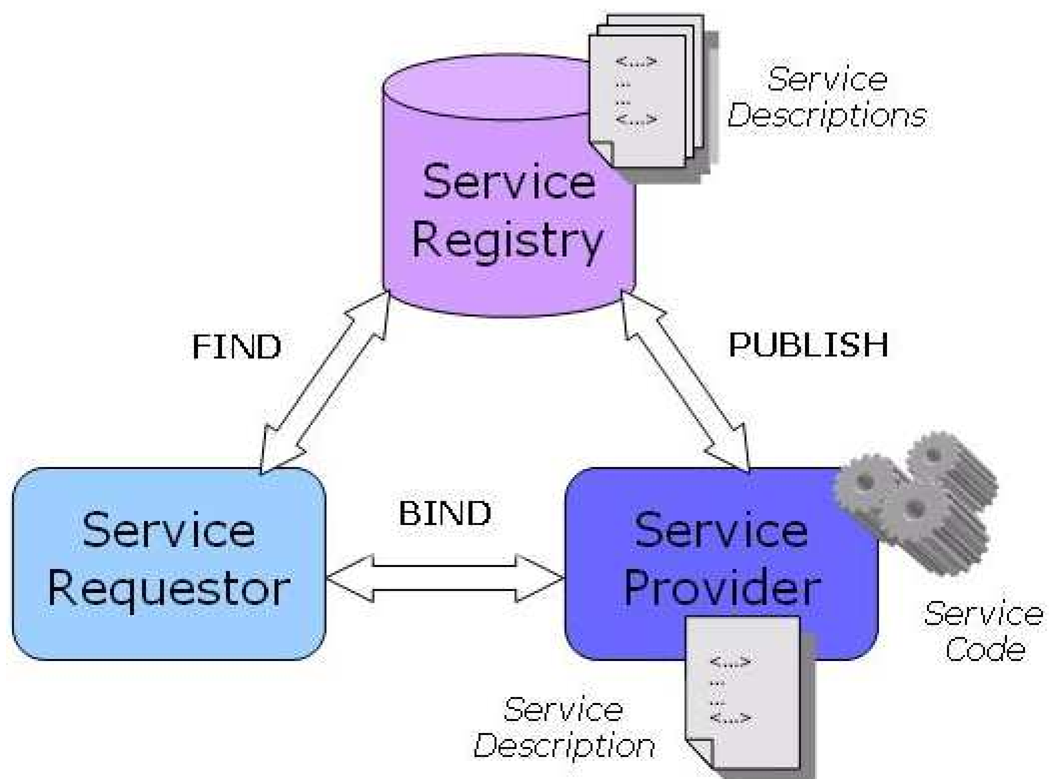

Figura 3.1: web service*: entità, operazioni, artefatti.*

3.2 Componenti dell'architettura

3.2.1 Entità

1. *service provider*:
   questa entità rappresenta il fornitore del servizio; dal punto di vista architetturale, rappresenta la piattaforma che ospita il servizio e ne gestisce l'accesso;

2. *service requestor*:
   questa entità rappresenta il fruitore del servizio offerto; dal punto di vista architetturale, questo ruolo viene svolto dall'applicazione che ricerca, invoca ed interagisce con il servizio. Il *service requestor* può essere un browser, oppure un'applicazione priva di interfaccia utente (ad es. un altro *web service*);

3. *service registry*:
   questa entità raccoglie tutte le descrizioni dei servizi pubblicate dai rispettivi *service provider*; grazie alle descrizioni dei servizi ottenute tramite il *service registry*, il *service requestor* rileva informazioni relative al collegamento con un certo servizio sia a tempo di sviluppo (*static binding*) che a tempo di esecuzione (*dynamic binding*). Se il collegamento tra il *service requestor* ed un certo servizio è statico, il ruolo del *service registry* risulta opzionale, poiché il *service provider* può direttamente inviare al *service requestor* la descrizione del servizio richiesta.

3.2.2 Operazioni

1. *publishing*:

   questa operazione rende un servizio accessibile dall'esterno attraverso la pubblicazione della sua *service description*, in modo che il *service requestor* possa cercarla e trovarla; il modo in cui la descrizione del servizio è pubblicata può variare a seconda dei requisiti dell'applicazione;

2. *finding*:

   con questa operazione, il *service requestor* ricerca ed ottiene la descrizione di un servizio direttamente dal *service provider* o richiedendola al *service registry*. Questa operazione può essere eseguita in due momenti diversi durante il ciclo di vita del *service requestor*: a tempo di sviluppo, per ricevere la descrizione dell'interfaccia del servizio, oppure a tempo di esecuzione, per ottenere le informazioni necessarie al collegamento con il servizio;

3. *binding*:

   grazie a questa operazione, il *service requestor* stabilisce un collegamento, invoca il servizio (*service*) ed inizia ad interagire con esso, a tempo di esecuzione, tramite i dettagli dedicati all'interno della descrizione del servizio.

3.2.3 Artefatti

1. *service*:
   rappresenta il software che implementa l'interfaccia collegata alla *service description*; il software in questione, ospitato dal *service provider*, deve essere distribuito su piattaforme in grado di supportare l'accesso alla rete;

2. *service description*:
   contiene i dettagli dell'interfaccia e dell'implementazione del servizio; più precisamente, ciò comprende i tipi di dato, le operazioni, le informazioni di collegamento e di indirizzamento di rete, necessari per interagire con il servizio stesso.

3.3 Ciclo di vita

Il ciclo di vita dei *web service* può essere suddiviso in 4 fasi (building, deployment, running, management) in ognuna delle quali possono agire le tre entità presenti nell'architettura.

1. *building*:
   questa fase comprende l'attività di sviluppo e quella di testing dell'implementazione del *web service*, nonché la definizione e la realizzazione della *service description*. Si possono scegliere diversi modi per implementare un *web service*: creandone uno nuovo, trasformando applicazioni esistenti

in *web service* e componendo *web service* e applicazioni già esistenti;

2. *deployment*:
   questa fase include la pubblicazione dell'interfaccia del servizio e della sua implementazione verso un *service requestor* o un *service registry*; inoltre è prevista un'attività di deploying del codice eseguibile del *web service* all'interno di un ambiente predisposto (tipicamente un Web Application Server come *Microsoft IIS*, *Apache Tomcat*, *Sun Application Server*, etc.).

3. *running*:
   durante questa fase il *web service* è attivo ed in grado di essere invocato; a questo punto gli eventuali *service requestor* possono eseguire sia operazioni di ricerca che di collegamento.

4. *management*:
   in questa fase si realizzano attività quali amministrazione, sicurezza, performance e qualità del servizio del *web service.*

3.4 *Stack* protocollare

Per poter eseguire le operazioni di pubblicazione, ricerca e collegamento garantendo l'interoperabilità, è necessario che l'architettura dei *web service* preveda uno *stack* protocollare che in-

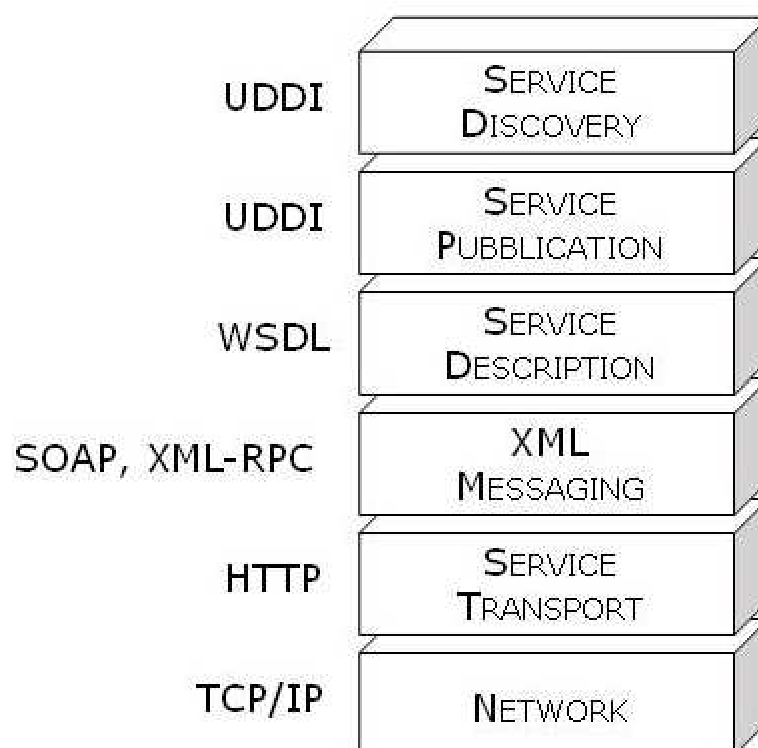

Figura 3.2: Stack protocollare dei web service.

cluda tecnologie standard ad ogni strato. Nella figura di seguito (figura 3.2) viene mostrato lo *stack* dei *web service*:

Come avviene per ogni architettura a livelli, gli strati superiori sono costruiti e svolgono le loro attività grazie alle funzionalità fornite dagli strati inferiori. Poiché i *web service* devono essere accessibili da remoto, alla base dello *stack* protocollare c'è lo strato di rete (Network), che coincide con il medesimo strato dello *stack TCP/IP*. Al livello superiore c'è lo strato etichettato come *service transport*, che si occupa del trasporto dei messaggi tra le applicazioni. Attualmente, a causa della sua massiccia affermazione, *HTTP* (*HyperText Transfer Protocol*) rappresenta il

protocollo standard nell'insieme dei protocolli di trasporto; oltre ad HTTP vi sono però altri protocolli che potrebbero costituire lo strato *service transport* dello *stack*, come *SMTP* (*Simple Mail Transfer Protocol*), *FTP* (*File Transfer Protocol*), *RMI/IIOP* (*Remote Method Invocation over Internet Inter Orb Protocol*), etc.

Tuttavia, quando si parla di *web service* in senso stretto, lo strato *service transport* è costituito dal protocollo *HTTP*; l'utilizzo di uno dei protocolli alternativi rappresenterebbe ugualmente un servizio accessibile da remoto, ma non un *web service.* Lo strato immediatamente superiore, *XML Messaging*, si occupa della codifica dei messaggi in un formato comune basato su una grammatica *XML* standard; allo stato attuale vi sono due protocolli che si contendono questo ruolo, *SOAP* (*Simple Object Access Protocol*) e *XML-RPC (XML for Remote Procedure Call*). *XML-RPC* è un protocollo molto semplice che usa lo scambio di messaggi *XML* per eseguire chiamate di procedura remota. Questo protocollo si basa su *HTTP*: le richieste sono codificate in *XML* ed inviate nel corpo di un *HTTP POST*, mentre le risposte vengono collocate all'interno del corpo di un *HTTP RESPONSE. XML-RPC* risulta spesso più semplice da utilizzare rispetto a *SOAP* ma, a differenza di quest'ultimo, non prevede l'uso di una grammatica standard per la descrizione del servizio. *SOAP* è un protocollo più flessibile di *XML-RPC* in quanto il suo funzionamento non è vincolato dall'impiego di un

particolare protocollo (come *HTTP*) e non è vincolato da un unico modello di comunicazione (RPC). Malgrado la sua duttilità, *SOAP* è abbinato all'uso di *HTTP* per costituire il cuore dei *web service*, e rappresenta lo standard per due tipi di interazione basati entrambi su *XML*: scambio di documenti (document-style) e chiamate di procedura remota (RPC-style). Mentre *XML-RPC* è semplicemente un'estensione di *HTTP* in grado di trattare il trasporto di messaggi *XML*, *SOAP* è un nuovo protocollo, indipendente da *HTTP*, che definisce una struttura complessa del messaggio *XML* trasportato. Lo strato successivo della pila protocollare dei *web service*, *service description*, si occupa appunto della descrizione dell'interfaccia di un particolare servizio e, attualmente è costituito da *WSDL* (*web service Description Language*). *WSDL* è una grammatica *XML* che rappresenta lo standard per descrivere l'interfaccia di un servizio offerto da un *web service*; questa interfaccia pubblica può includere informazioni sulle funzionalità rese disponibili dal servizio, sui tipi di dato per lo scambio dei messaggi *XML*, sullo specifico protocollo di trasporto da utilizzare per l'operazione di *binding* con il *web service*, etc. *WSDL* non è vincolato ad un particolare protocollo di messaggistica *XML*, ma include un'estensione nativa per la descrizione ed il trattamento di servizi *SOAP*. I primi quattro strati della pila protocollare forniscono la base per lo sviluppo e l'uso di qualsiasi servizio remoto; la situazione più pratica prevede la scelta di *HTTP* come protocollo dedicato al

trasporto dei messaggi *XML*, *SOAP* come protocollo per la codifica dei messaggi in formato *XML*, e *WSDL* per la descrizione delle interfacce di servizio esposte. Questi tre protocolli standard costituiscono la base per l'interoperabilità dei *web service.* I successivi due strati dello *stack* possono essere implementati con uno spettro di scelte più ampio. Qualsiasi azione renda disponibile un documento *WSDL* verso un potenziale *service requestor*, viene indicata come *service publishing* (pubblicazione del servizio). L'esempio più semplice di pubblicazione di servizio è quella che viene definita pubblicazione diretta: in questo caso il *service provider* si occupa direttamente di inviare il documento *WSDL* al *service requestor*, ad esempio tramite e-mail. In alternativa, il *service provider* può pubblicare il documento *WSDL*, che descrive l'interfaccia del servizio offerto, verso un *WSDL registry* locale oppure verso un UDDI registry. Poiché un *web service* non può essere ricercato e scoperto prima della sua pubblicazione, lo strato *service discovery* dipende dallo strato sottostante *service pubblication.* Le diverse modalità con cui un *web service* può essere ricercato sono dipendenti dai meccanismi di pubblicazione di servizio. Qualsiasi azione consenta al *service requestor* di accedere alla descrizione di un servizio e la renda disponibile all'applicazione che ne fa uso a tempo di esecuzione, viene detta service discovery (ricerca del servizio). L'esempio più semplice di ricerca di servizio è la ricerca statica, in cui il *service requestor* ottiene il documento *WSDL* da un file locale,

spesso inviato dal *service provider* attraverso una pubblicazione diretta. In alternativa il servizio può essere scoperto a tempo di sviluppo o di esecuzione, usando un *WSDL registry* locale o un UDDI registry.

3.5 Protocolli specifici

Nei tre paragrafi seguenti andremo a descrivere in dettaglio i tre protocolli fondamentali che costituiscono lo *stack* dei *web service.*

3.5.1 *SOAP*

Una delle caratteristiche fondamentali che rendono i *web service* interoperabili indipendentemente dall'ambiente in cui sono sviluppati, è l'uso di *XML* come codifica standard per lo scambio di messaggi. *SOAP*[1] è il protocollo che rappresenta lo standard attuale per lo scambio di messaggi in formato *XML* tra applicazioni distribuite. *SOAP* è *XML*, ossia è un'applicazione particolare delle specifiche fornite da *XML*, che si basa molto sugli standard derivati da esso, come ad esempio *XML-Schema* e *XML-Namespace.* Le specifiche *SOAP* definiscono 3 parti principali:

- *SOAP envelope specification*:

[1] http://www.w3.org/TR/2003/REC?soap12?part0?20030624

rappresenta un involucro *XML* che definisce le regole per il trasferimento dei messaggi tra le applicazioni;

- *data encoding rules*:
  comprende le regole per la codifica di tipi di dato particolari, che consentono alle applicazioni che ne fanno uso, di interpretarli correttamente indipendentemente dalla piattaforma su cui sono sviluppate;
- *RPC conventions*:
  fornisce i meccanismi necessari per gestire un modello di comunicazione basato su chiamate di procedura remota e risposte.

Paradigmi di comunicazione: RPC e EDI

Lo scambio di messaggi *XML*, e quindi il protocollo *SOAP*, ha due contesti applicativi, RPC e EDI. RPC (Remote Procedure Call) rappresenta il modello di interazione di base in ambito distribuito; tramite il modello RPC, un programma può invocare una procedura (metodo, in gergo Object-Oriented) esposta da un altro programma, attraverso un meccanismo per il passaggio dei parametri e per la ricezione di eventuali valori di ritorno. EDI (Electonic Document Interchange) rappresenta il modello di intrerazione tipico per le transazioni commerciali, che definisce il formato e l'interpretazione standard per i documenti finanziari. Se *SOAP* viene abbinato al contesto RPC, si ottiene il modello *SOAP* RPC-style e il documento *XML* corrispondente

rappresenterà i parametri ed i valori di ritorno relativi ad una particolare procedura remota. Se *SOAP* viene utilizzato in un contesto EDI, si ottiene il cosiddetto modello *SOAP document-style*, e il documento *XML* rappresenterà, ad esempio, un ordine di acquisto.

Formato del messaggio

Ogni messaggio *SOAP* è costituito da un *envelope* (involucro), rappresentato dall'elemento *Envelope*, che contiene uno header (elemento Header opzionale) ed un *body* (elemento *Body* obbligatorio). Lo *header* contiene blocchi di informazione per il trattamento del messaggio, come ad esempio i parametri relativi al routing, quelli relativi all'autorizzazione e autenticazione e quelli dedicati alle transazioni. Il body contiene il messaggio da trasportare: qualsiasi cosa possa essere espressa con una sintassi *XML*, può costituire il *body* di un messaggio *SOAP*. La figura 3.3 mostra il formato del messaggio *SOAP 1.2*.

Regole di codifica dei dati

Le specifiche *SOAP* prevedono un insieme di regole per la codifica di alcuni tipi di dato; queste regole consentono al protocollo di gestire tipi di dato come *integer*, *float*, *double*, *array*, ed essendo implementate direttamente dal *toolkit SOAP* usato, sono totalmente nascoste allo sviluppatore dell'applicazione. Nonostante la maggior parte delle applicazioni non debbano preoccuparsi

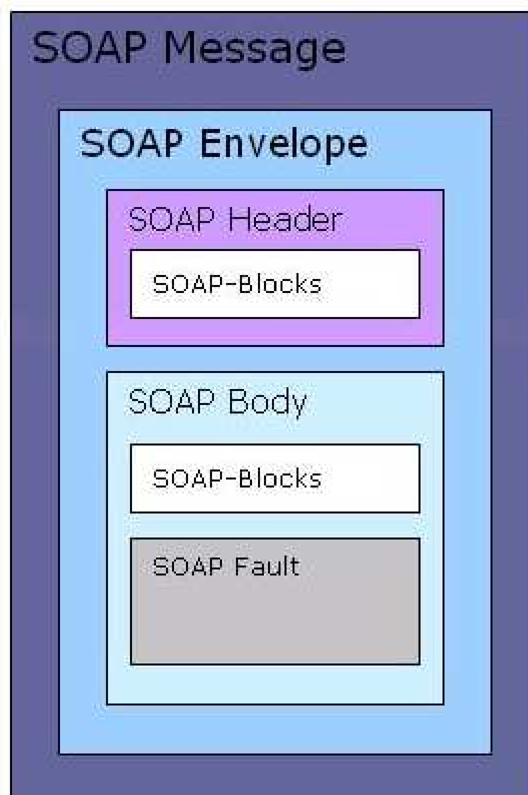

Figura 3.3: Il formato del messaggio SOAP.

delle regole di codifica del protocollo, vi sono casi in cui è necessario manipolare direttamente il messaggio *SOAP*. Le regole di codifica comprese nelle specifiche *SOAP* non sono da intendersi obbligatorie, ma sono comunque promosse ed incoraggiate dal *W3C* (*World Wide Web Consortium*). Questa libertà di azione consente di scegliere schemi di codifica diversi a seconda delle necessità che si presentano. Le specifiche *SOAP* adottano le stesse convenzioni proposte per la codifica dei tipi di dato di un generico documento *XML*; tali convenzioni sono state rilasciate dal *W3C* all'interno delle specifiche *XML-Schema Part II*. Malgrado *SOAP* adotti le stesse regole proposte da *XML-Schema* per i tipi di dato predefiniti, mantiene alcune convenzioni proprie per i tipi di dato non standardizzati da *XML-Schema*, come gli array e i puntatori. I tipi di dato *SOAP* si dividono in 2 categorie: tipi scalari e tipi composti. I tipi scalari contengono esattamente un valore, mentre i tipi composti contengono valori multipli e sono a loro volta suddivisi in array e strutture.

***SOAP* via *HTTP* (RPC-style)**

Come già detto, *SOAP* non è vincolato all'uso di un particolare protocollo di trasporto; ciononostante, le specifiche *SOAP* includono i dettagli solamente per il trasporto via *HTTP*, e perciò *HTTP* resta il più importante protocollo per la trasmissione di messaggi *SOAP*. Per una naturale analogia, le richieste *SOAP* vengono inviate attraverso richieste *HTTP* e, allo stesso modo,

le risposte *SOAP* sono contenute in risposte *HTTP*. Le richieste *SOAP* possono essere trasportate sia come *HTTP GET* che come *HTTP POST*, anche se quest'ultima possibilità è da preferire poiché molti Web Server pongono un limite sulla lunghezza della stringa che rappresenta la richiesta di tipo *HTTP GET*. Sia le richieste che le risposte *HTTP* devono specificare come *text/xml* il tipo di contenuto del messaggio.

SOAP e il *W3C*

SOAP 1.1 è stato esaminato dal *W3C* per la prima volta nel maggio 2000, a fronte della richiesta di importanti aziende come *Microsoft*, *IBM*, *Ariba*, etc. Nel settembre del 2000, il *W3C* ha istituito un nuovo gruppo di lavoro, *XML protocol working group*, il cui obiettivo era quello di definire un protocollo standard basato su *XML* per lo scambio di informazioni. A luglio del 2001, *XML protocol working group* ha rilasciato una bozza di *SOAP 1.2*, attualmente in fase di sviluppo e miglioramento. Il *W3C* ha suddiviso le specifiche di *SOAP 1.2* in 2 parti: *Part I* descrive la struttura e l'involucro dei messaggi *SOAP*, mentre *Part II* definisce le regole di codifica, le convenzioni per l'interazione RPC e alcuni dettagli relativi a *SOAP* via *HTTP*.

3.5.2 WSDL (*web service Description Language*)

WSDL[2] rappresenta un insieme di specifiche attraverso cui è possibile descrivere i *web service* utilizzando una grammatica *XML* comune e standard. In particolare si occupa di 4 aspetti:

- informazioni relative all'interfaccia del *web service* e descrizione di tutte le funzioni disponibili; item informazioni sui tipi di dato da trasferire, sia nei messaggi di richiesta che in quelli di eventuale risposta;
- informazioni di collegamento riguardo al protocollo di trasporto usato;
- informazioni di indirizzamento per la localizzazione del *web service*.

WSDL costituisce una sorta di contratto tra il *service requestor* ed il *service provider* indipendente dalla piattaforma e dal linguaggio, ed è usato principalmente per descrivere servizi *SOAP*.

3.5.3 WSDL Specification

Le specifiche *WSDL* sono suddivise in 6 elementi principali:

- *definitions*:
  l'elemento definitions costituisce la radice di ogni documento *WSDL*, indica il nome del *web service* e dichiara i namespaces utilizzati nel documento;

[2] http://www.w3.org/TR/2004/WD-wsdl20-20040326

- *types*:
  l'elemento types descrive tutti i tipi di dato usati tra il client ed il server; *WSDL* non è vincolato dall'utilizzo di un particolare sistema di tipi, tuttavia utilizza quello dichiarato nelle specifiche *XML-Schema* del *W3C*. Se il servizio che si vuole descrivere utilizza solo tipi di dato conformi a questo standard, ad esempio interi o stringhe, l'elemento types risulta superfluo;

- *message*:
  questo elemento descrive un messaggio scambiato tra il client ed il server e può rappresentare sia una richiesta che una risposta; specifica il nome del messaggio ed eventuali parametri di input e di output attraverso elementi part;

- *portType*:
  Questo elemento raggruppa più elementi message per costituire un'interazione completa tra client e server; ad esempio, un elemento portType può essere costituito da un elemento message relativo ad una richiesta ed uno relativo alla risposta di uno schema *SOAP-RPC*;

- *binding*:
  L'elemento *binding* descrive le modalità effettive con cui il servizio viene implementato; *WSDL* include un supporto nativo per i servizi implementati tramite *SOAP*;

- *service*:

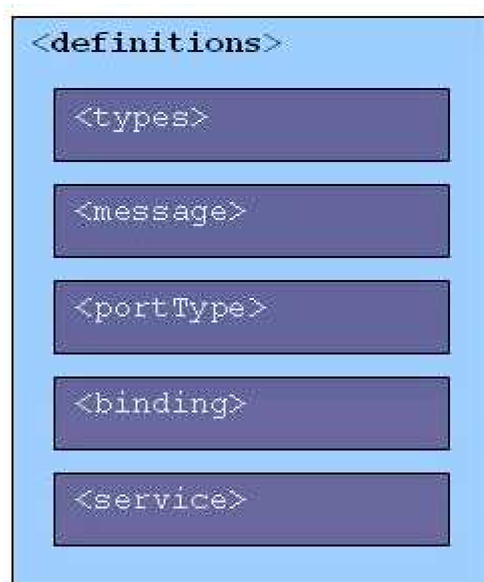

Figura 3.4: Struttura del documento WSDL.

l'elemento service specifica l'indirizzo per invocare un determinato servizio; quasi sempre questo consiste in una URL relativa ad un servizio *SOAP*.

La figura seguente (figura 3.4) mostra la struttura del generico documento *WSDL*.

Oltre a questi 6 elementi, le specifiche *WSDL* ne definiscono altri 2:

- *documentation*:
  fornisce una documentazione human-readable del servizio;

- *import*:
  specifica documenti *WSDL* o *XML-Schema* addizionali da importare.

WSDL* e il *W3C

WSDL 1.1 è stato sottoposto al *W3C* nel marzo del 2001 da aziende come *IBM*, *Microsoft* e *Ariba*. All'interno del *W3C* si è creato un gruppo di lavoro nell'ambito della *web service activity*, denominato *web service description working group*, che si occupa delle specifiche di *WSDL 2.0*.

3.5.4 *UDDI* (*Universal Description Discovery and Integration*)

UDDI[3] è stato promosso da *Microsoft* ed *IBM* nel maggio 2001; fornisce un insieme di meccanismi per la pubblicazione, la ricerca e l'integrazione di qualsiasi servizio accessibile da remoto, e in particolare dei *web service*. Consiste sostanzialmente di due parti: la prima è costituita da un insieme di specifiche per la costruzione di un archivio distribuito che raccolga informazioni relative a servizi web. I dati sono raccolti e codificati secondo un formato *XML* specifico e le *API UDDI* consentono sia di pubblicare nuovi dati, sia di ricercare dati esistenti conformi alle specifiche. La seconda parte, detta *UDDI Business Registry*, rappresenta un'implementazione completa delle specifiche *UDDI*. *UDDI* non contiene solamente servizi web basati su *SOAP*, ma può essere utilizzato per raccogliere informazioni relative a qualsiasi servizio accessibile da remoto (CORBA,

[3]http://www.uddi.org

RMI/IIOP, etc.). I dati raccolti da *UDDI* si possono suddividere in 3 categorie principali:

- *white pages*:
  Includono informazioni generali relative ad una particolare compagnia che decide di pubblicare un servizio web, quali ad esempio il nome della compagnia stessa, una sua descrizione e le informazioni di contatto;

- *yellow pages*:
  Includono informazioni relative sia alla compagnia sia ai servizi che essa offre, come ad esempio codici di prodotto basati su tassonomie standard;

- *green pages*:
  Includono le informazioni tecniche di un particolare *web service*, che possono comprendere ad esempio un riferimento per la localizzazione del *web service*;

L'architettura *UDDI* consiste di 3 parti: *UDDI data model*, *UDDI API* e *UDDI cloud services*.

- *UDDI data model*
  Il modello dei dati utilizzato da *UDDI* è rappresentato da due *XML-Schema* (versione 1.0[4] e 2.0[5]). Questo schema identifica 4 categorie di informazioni che corrispondono ad altrettanti elementi *XML*:

[4]http://www.uddi.org/schema/2001/uddi_v1.xsd
[5]http://www.uddi.org/schema/uddi_v2.xsd

  - *businessEntity*:
    contiene informazioni relative al fornitore del servizio (nome, contatti, etc.);
  - *businessService*:
    include informazioni di un singolo *web service* o di un gruppo di *web service* collegati (nome, descrizione, *bindingTemplate*);
  - *bindingTemplate*:
    specifica dove e come poter accedere ad un particolare *web service*;
  - *tModel*:
    include descrizioni e riferimenti a specifiche tecniche esterne al *web service*.

- *UDDI API*
  Si tratta di un insieme di API dedicate al protocollo *SOAP* per la pubblicazione e la ricerca di dati sul registry UDDI.

- *UDDI cloud services*
  Questi elementi identificano i siti web di operatori che forniscono un'implementazione delle specifiche *UDDI* (ad es. http://test.uddi.microsoft.com).

Capitolo 4

Impostazione del problema

L'obiettivo principale proposto da raggiungere è quello di studiare un'architettura che permetta di risolvere il problema della cooperazione tra dispositivi domotici eterogenei dal punto di vista tecnologico e di implementare un'applicazione basata su di essa. L'architettura usata deve inoltre permettere il controllo dei dispositivi domotici in remoto, indipendentemente dalla loro locazione.

Questo diventa possibile creando un livello di astrazione che permetta di descrivere i dispositivi domotici sia dal punto di vista fisico che comportamentale in modo di avere una visione omogenea indipendentemente dalle tecnologie sottostanti.

L'idea è quella di avere un gruppo di *web service* ognuno dei quali capace di interagire con i dispositivi domotici da lui fisicamente raggiungibili, tipicamente in un'area geografica ben precisa circoscritta intorno sua locazione fisica. Ogni area geografica, rappresentata logicamente dal *web service*, racchiude come ser-

vizi l'insieme delle funzionalità offerte dai dispositivi presenti. Ogni *web service* deve essere in grado di interpretare e comportarsi conseguentemente allo stato del livello astratto usando dei moduli capaci di interfacciarsi con i *middleware* dei dispositivi.

4.1 Scenario

Per comprendere la necessità di un *framework* che garantisse l'interoperabilità tra i diversi sistemi domotici presenti sul mercato, siamo partiti da uno scenario generico. In un ipotetico ambiente in cui coesistono alcuni tra i principali middleware domotici, è possibile identificare ogni sistema come una particolare "sottorete", pur senza soffermarsi sui dettagli e sulle infrastrutture necessarie. La Figura 4.1 mostra lo scenario appena descritto:

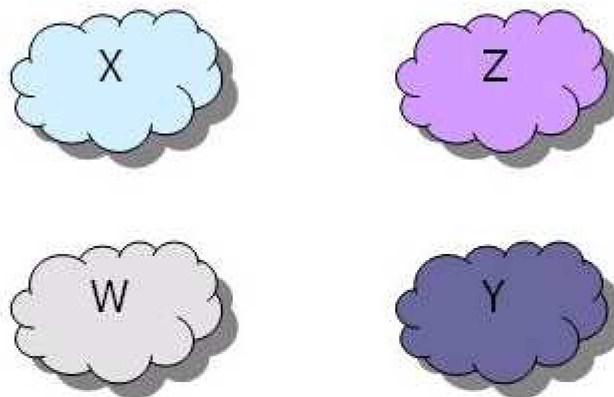

Figura 4.1: Rappresentazione dei middleware domotici.

Ogni "nuvola" rappresenta un *middleware*, ovvero una sot-

torete all'interno della quale sono presenti sia le infrastrutture *hardware* e *software* per il supporto, sia i dispositivi conformi.

Tutti i dispositivi all'interno della stessa sottorete colloquiano e cooperano per costituire un sistema funzionante ed indipendente, ma chiuso. Di fatto è impossibile controllare un dispositivo che si trova in una nuvola tramite un altro presente in una nuvola diversa, poiché non esiste nessun tipo di collegamento tra le due sottoreti. Per collegamento si intende un'infrastruttura logica che consenta ai dispositivi di qualsiasi sottorete di conoscere tutti i dispositivi presenti nelle altre sottoreti.

Infine, anche qualora fosse possibile avere una visione completa dell'intera rete, sarebbe comunque necessario un meccanismo comune per lo scambio di informazioni tra i dispositivi.

Vi sono quindi due aspetti da risolvere: da un lato riuscire a costruire un'infrastruttura logica di collegamento tra i vari middleware, dall'altro mettere a punto un meccanismo di comunicazione universale veicolato da quest'infrastruttura.

Una possibile soluzione a questi due problemi è quella di introdurre tra ogni coppia di sottoreti un modulo hardware e/o software che sia provvisto di un'interfaccia verso entrambe le nuvole.

Ogni modulo, che nella terminologia generale è detto *gateway*, deve non solo fornire funzionalità di traduzione tra i due protocolli delle sottoreti che collega, ma anche conciliare le differenze tra i paradigmi di comunicazione su cui si basano i due sistemi.

Quest'ultima necessità risulta evidente quando i sistemi domotici che si desidera collegare presentano caratteristiche notevolmente differenti (ad es. *middleware* a configurazione manuale vs. *plug and play*). L'implementazione di un particolare gateway non può dunque prescindere da una conoscenza approfondita di entrambi i sistemi domotici da collegare.

Questa soluzione, tuttavia, risulterebbe poco scalabile a causa del numero di gateway necessari al crescere dei sottosistemi da collegare: infatti, all'aumentare del numero delle sottoreti, il numero di gateway da sviluppare cresce sensibilmente.

A questa prima soluzione ne abbiamo preferita una seconda, che interviene ad un livello superiore.

Per riconoscere la validità di questo approccio e la sua applicabilità, siamo partiti da un'analogia che deriva direttamente dal mondo delle reti di calcolatori.

In passato, molti costruttori hanno sviluppato infrastrutture ad *hoc* per la costituzione di reti che permettessero la comunicazione tra piattaforme e periferiche dello stesso marchio; si pensi ad esempio alla rete *AppleTalk* dedicata ai sistemi *Macintosh*, a quella *NFS* per i sistemi *Unix*, a *NetBeui* per i PC *Windows* oppure alle reti geografiche *IBM SNA*, *NOVELL/IPX*, etc. Queste soluzioni limitavano notevolmente la potenziale espansione delle reti poiché non permettevano la comunicazione tra macchine che supportavano infrastrutture di rete diverse.

Proprio per questo motivo, vi sono stati tentativi da parte di

singole aziende, peraltro falliti, di imporre la propria infrastruttura come unico standard, come nel caso di *Microsoft* al rilascio del sistema operativo *Windows 95*.

Di fatto nessuna infrastruttura ha mai raggiunto un'egemonia sulle altre, principalmente a motivo dello sviluppo di un'infrastruttura innovativa in grado di superare il vincolo della unicità di piattaforma: questa infrastruttura, di ispirazione *ISO/OSI*, è il noto *stack* di protocolli *TCP/IP*. Seguendo lo stesso approccio in contesto domotico, si è pensato di introdurre un'ulteriore nuvola atta a gestire i diversi sistemi domotici al di là delle differenze fra essi, permettendo così la comunicazione tra nodi appartenenti a sottoreti distinte.

Questa nuova infrastruttura, denominata *domoNet*, si basa sui *web service* e fornisce una visione dell'intera topologia della rete secondo un modello *SOA*.

Per garantire l'interoperabilità tra nodi comunicanti attraverso l'infrastruttura *domoNet*, è necessario sviluppare, oltre ad opportuni *gateway*, una grammatica *XML* standard detta *domoML*.

Rispetto alla prima soluzione proposta, questa è certamente più scalabile, in quanto richiede solamente un *gateway* per ogni sistema domotico da collegare a *domoNet*.

Ogni *gateway*, che d'ora in avanti chiameremo *tech manager*, dovrà possedere da un lato un'interfaccia verso il particolare sistema domotico a cui si riferisce, e dall'altro un'interfaccia verso

domoNet (figura 4.2).

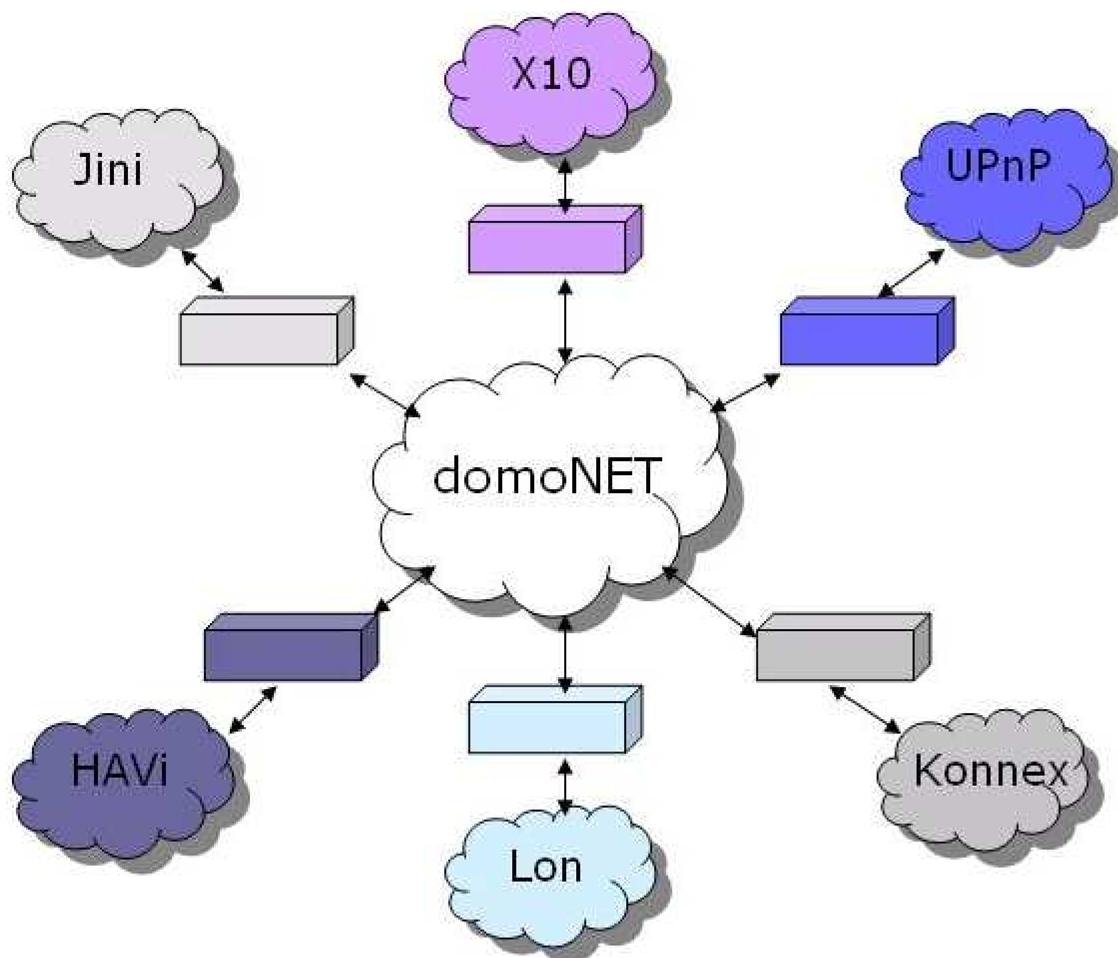

Figura 4.2: Il framework domoNet.

4.2 Un primo sguardo all'architettura domoNet

L'architettura *domoNet* è composta da un insieme di server (*web service*) e da dei possibili *client* per il controllo remoto dei dispositivi domotici. Il controllo remoto permette di interagire

con tutti i dispositivi raggiungibili dai *web service* appartenenti alla rete *domoNet*.

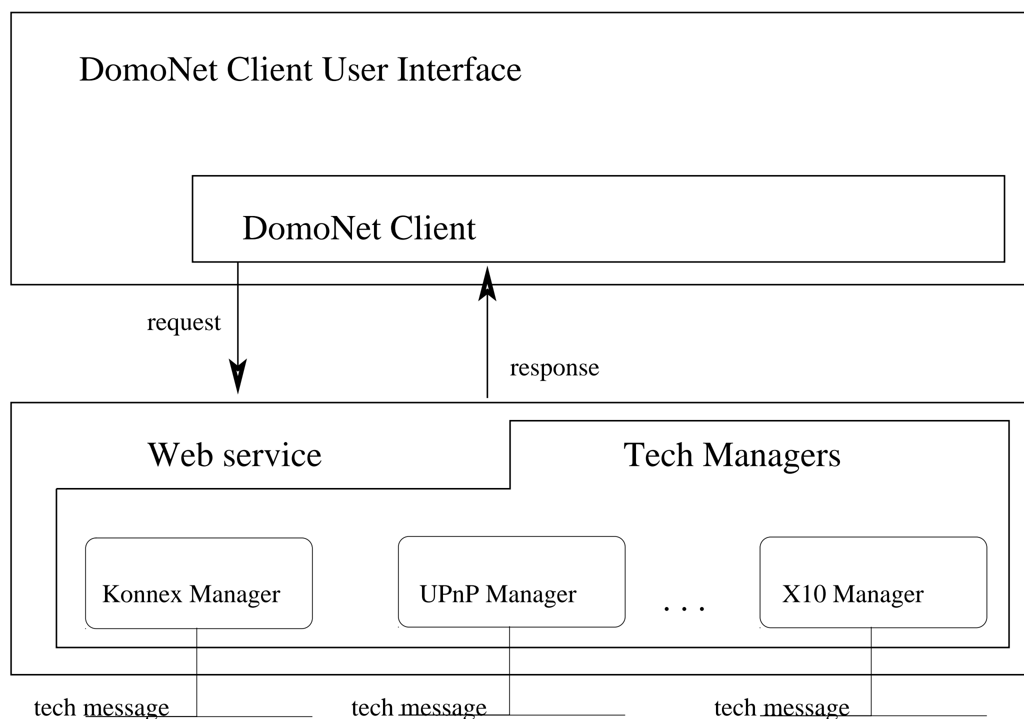

Figura 4.3: L'architettura domoNet

In figura 4.3 viene dato un primo sguardo all'architettura *domoNet*. L'architettura mostrata è composta da una parte *client* (il *domoNetClient* + *domoNetClientUI*) che, connessa alla parte *server* (*web service* + *tech mangers*) è in grado di effettuare richieste tramite *TCP/IP* affinché vengano eseguite operazioni sui dispositivi. La parte server, eseguita la richiesta, è in grado di rispondere al *client* con un dato o con un codice interno di successo o fallimento, in base alla riuscita o meno dell'operazione.

Il livello astratto è implementato attraverso un linguaggio

basato su *XML* chiamato *domoML*. Con questo linguaggio è possibile descrivere tutti i dispositivi domotici (*domoDevice*) e i relativi messaggi (*domoMessage*). *DomoML* è un linguaggio intermedio da e verso il quale tradurre descrizioni e azioni.

Ogni *web service* è connesso fisicamente ai dispositivi domotici ed è in grado di controllarli invocando moduli specializzati, chiamati *tech manager*, in base alla tecnologia con la quale è necessario interagire.

Per implementare la cooperazione, ogni *web service* è in grado di catturare un messaggio proveniente dal modulo di una tecnologia, convertirlo in *domoMessage* per poi indirizzarlo al modulo gestore della tecnologia di destinazione che provvederà a riconvertirlo in modo che possa essere eseguito.

Per implementare il controllo remoto, il *client* è in grado di avere una lista dei dispositivi domotici connessi ad uno o più *web service*. Ogni dispositivo è rappresentato usando il formalismo del *domoDevice*. Al momento in cui deve essere eseguita un'azione remota, il client genera il corrispondente *domoMessage* e lo invia verso il giusto *web service*. Raggiunto il *web service*, il *domoMessage* viene inviato al giusto modulo ed eseguito. L'esito o l'eventuale risposta derivata dall'esecuzione viene ritrasmessa al mittente della richiesta col formalismo *domoMessage*.

4.3 Terminologia

- *dispositivo*: apparecchiatura fisica appartenente ad una determinata tecnologia;
- *domoML*: formalismo *XML* usato per descrivere e implementare le funzionalità dei dispositivi nell'applicazione;
- *domoDevice*: un dispositivo *domoNet* descritto tramite il linguaggio *domoML*;
- *domoMessage*: descrizione di un messaggio tramite il linguaggio *domoML*;
- *domoAddress*: una coppia composta dall'*URL* del *web service* e dall'*id* che identifica in maniera univoca il *domoDevice*;
- *real address*: l'indirizzo reale di un *device* nella tecnologia di appartenenza;
- *tech manager*: modulo del *web service* che amministra una tecnologia. Ogni *web service* può avere più *tech manager* di tecnologie differenti.
- *tech message*: messaggio che proviene o è generato da un *tech manager* o da un client esterno al *web service.*

Capitolo 5

DomoML

DomoML è un un linguaggio per rappresentare in maniera compatta i dispositivi domotici: i relativi servizi offerti e le relative interazioni, astraendo dalla tecnologia di appartenenza.

DomoML, usato con questa architettura, è un *middle language* da e verso quale tradurre le rappresentazioni dei dispositivi e dei pacchetti. Tutta la parte logica dell'architettura usa *domoML* e solamente le interazioni fisiche con i dispositivi per costruire i corrispondenti *domoDevice* e per l'esecuzione dei servizi, all'interno dei moduli (*tech manager*), usano il linguaggio specifico proprio di ogni tecnologia.

DomoML si occupa attraverso i *tag*:

- *domoDevice*: alla descrizione astratta dei dispositivi fatte secondo *domoML*;
- *domoMessage*: alla descrizione astratta delle interazioni da e verso i dispositivi fatte secondo *domoML*.

5.1 *DomoDevice*

Il *domoDevice* ha lo scopo di creare un meccanismo semplice, compatto ed efficace per rappresentare i dispositivi astraendo dalla tecnologie.

Un *domoDevice* può essere rappresentato attraverso un albero largo e poco profondo (la grammatica è altamente attributata e con pochi figli). L'albero ha al massimo 4 livelli:

1. `device`:
   apre il tag che descrive il *domoDevice* dando generiche informazioni come:

   - `description`:
     una descrizione in lingua naturale del *domoDevice*;
   - `id`:
     valore usato per prendere e generare il `domoDeviceId`. Identifica il *domoDevice* all'interno del *web service*;
   - `manufacturer`:
     il produttore del dispositivo;
   - `positionDescription`:
     una descrizione in lingua naturale della locazione del dispositivo;
   - `serialNumber`:
     il numero seriale (può essere composto anche da lettere) del dispositivo;

- `tech`:
  la tecnologia originale del *domoDevice* rappresentato;
- `type`:
  la tipologia del dispositivo;
- `URL`:
  usato per prendere e generare il `domoDeviceId`. Identifica il *web service* che tiene di dispositivo.

Tutti i campi sono opzionali eccetto per l'`id`, `URL` e `tech` perché permettono di identificare e usare il dispositivo.

2. `service`:
   descrive un singolo servizio offerto dal *domoDevice*. Questa informazione è usata per creare un *domoMessage* e per convertirlo nel *tech message*. Per ogni *domoDevice* ci possono essere più servizi. I campi di questo tag sono:

   - `description`:
     una descrizione in linguaggio naturale del servizio;
   - `name`:
     un identificatore che può essere utile quando il *domoMessage* viene generato e tradotto in *tech message*.
   - `output`:
     il `domoML.domoDevice.DomoDevice.DataType` si aspetta un valore di ritorno quando viene eseguito il *domoMessage*. Questo campo non va messo se non è previsto alcun valore di ritorno;

- `outputDescription`:
  una descrizione in linguaggio naturare per l'attributo `output`. Se non è presente l'attributo `output` neanche questo attributo deve essere messo;
- `prettyName`:
  una etichetta in in linguaggio naturale del servizio;

3. due possibili tag per questo livello:

   - `input`:
     indica che il servizio ha bisogno di un valore di input per poter essere eseguito. Questo tag è opzionale e ogni servizio può avere più input. I campi per questo tag sono:
     - `name`: un identificativo dell'input;
     - `description`: una descrizione dell'input;
     - `type`: il `domoML.domoDevice.DomoDevice.DataType` dell'input atteso;
   - `linkedService`:
     il servizio da associare quando questo servizio viene eseguito. Può essere invocato un servizio dello stesso o di un qualsiasi altro *domoDevice*. L'associazione di serivizi non ha vincoli semantici ovvero è possibile combinare un servizio con qualsiasi altro servizio a patto di riuscire a dare dei valori ragionevoli ai `linkedInput` (discusso successivamente). Questo *tag* è il cuore della

tesi in quanto permette di risolvere il problema della cooperazione tra dispositivi reali eterogenei. Questo tag è opzionale. I suoi attributi sono:

- `URL`: l'identificativo del *web service* che gestisce il *domoDevice* da collegare al servizio;
- `id`: l'identificativo del *domoDevice* da collegare al servizio all'interno del *web service*;
- `service`: il nome del servizio del *domoDevice* individuato dai precedenti attributi da collegare;

4. in base al tag precedente, a questo livello è possibile avere:

   - `allowed`:
     se il tag precedente è `input`. Rappresenta un possibile valore che può assumere l'input. Questo parametro è opzionale e un `input` può avere più `allowed`; L'unico attributo per questo tag è:
     - `value`: il valore in formato stringa ammesso.
   - `linkedInput`:
     se il *tag* precedente è `linkedService`. Rappresenta l'associazione di un *input* del presente servizio con un *input* del servizio da richiamare. Ogni *input* del servizio da richiamare deve avere un'associazione. In questo modo, il valore dell'input del presente servizio viene passato come *input* al servizio da richiamare; Gli attributi per questo tag sono:

  - `from`: il nome dell'input dal quale prendere il valore;
  - `to`: il nome dell'input che deve usare il valore preso;

Un esempio di *domoDevice* per una semplice lampadina è:

```
<device description="lampada a risparimo energetico"
 id="0" manufacturer="pholips"
 postionDescription="comodino accanto al letto"
 serialNumber="xxxxxxxxx" tech="KNX" type="lampada"
 url="http://www.questowebservice.it/servizio">
    <service description="Get the status"
     output="BOOLEAN" outputDescription="The value"
     name="GET_STATUS" prettyName="Get status" />
    <service description="Set the status"
     name="SET_STATUS" prettyName="Set status">
        <input description="The value" name="status"
         type="BOOLEAN">
            <allowed value="TRUE" />
            <allowed value="FALSE" />
        </input>
<linkedService id="3" service="setPower"
         url="http://www.altrowebservice.it/servizio">
            <linkedInput from="status" to="power" />
        </linkedService>
    </service>
</device>
```

In questo esempio è possibile vedere che il *domoDevice* descritto è una lampadina a risparmio energetico prodotta dalla ditta `pholips` e che si trova sul comodino accanto al letto. Usa la tecnologia *Konnex* e fa parte della categoria `lampada`. La lampada è comandata dal *web service*
`http://www.questowebservice.it/servizio`.

I servizi offerti dalla lampada sono due: prendere il suo stato corrente in formato booleano e settare il suo stato attraverso un input, sempre booleano, che può assumere i valori `TRUE` o `FALSE`. Quando viene invocato quest'ultimo servizio, ne viene chiamato anche un altro di nome `setPower` appartenente al *domoDevice* gestito dal *web service*
`http://www.altrowebservice.it/servizio` con `id` 3. Il valore dell'input assegnato al primo servizio (`status`) viene assegnato anche al secondo (`power`).

Se viene invocato quindi il servizio *setStatus* del *domoDevice* con
`url="http://www.questowebservice.it/servizio"` e con `id=0` passando come valore di input `TRUE`, viene anche invocato il servizio `setPower` del *domoDevice*
`url="http://www.altrowebservice.it/servizio"`, con `id=3` e con valore di input `TRUE`.

5.2 *DomoMessage*

Il *domoMessage* ha lo scopo di creare un meccanismo semplice, compatto ed efficace per rappresentare le interazioni tra i *domoDevice* astraendo dalle tecnologie.

Anche il *domoMessage* può essere rappresentato attraverso un albero largo e poco profondo (si usa una grammatica altamente attributata e con pochi figli). L'albero ha al massimo 2 livelli:

1. `message`:
   apre il tag che descrive il *domoMessage* e contiene i seguenti attributi:

   - `message`:
     il corpo del messaggio, tipicamente il nome del servizio da invocare;
   - `messageType`:
     il tipo di messaggio. Può essere:
     - `COMMAND`: se trattasi di servizio da eseguire;
     - `SUCCESS`: se richiesto l'esito successivo al `COMMAND`. In questo caso il campo `message` può contenere il valore di risposta dell'esecuzione del servizio;
     - `FAILURE`: se richiesto l'esito successivo al `COMMAND`. In questo caso il campo `message` può contenere il codice di errore del fallimento;

- `receiverId`:
  l'identificatore del *domoDevice* ricevente del messaggio;
- `receiverURL`:
  l'url del *web service* che contiene il *domoDevice* richiesto;
- `senderId`:
  l'identificatore del *domoDevice* mittente del messaggio;
- `senderURL`:
  l'url del *web service* che contiene il *domoDevice* che invia il messaggio;

2. `input`:
   i valori di input da associare al `message`. Questo tag è opzionale e ogni tag `message` può avere più tag `input`. Gli attributi per questo tag sono:
   - `name`: il nome dell'input;
   - `type`: il `DomoDevice.DataType` dell'input;
   - `value`: il valore in formato stringa dell'input;

Un esempio di *domoMessage* per accendere una semplice lampadina è:

```
<message message="SET_STATUS" messageType="COMMAND"
 receiverId="1"
 receiverURL="http://www.altrowebservice.it/servizio"
 senderId="0"
```

```
 senderURL="http://www.questowebservice.it/servizio">
    <input name="status" type="BOOLEAN"
     value="TRUE" />
</message>
```

In questo esempio viene richisto di eseguire il servizio SET_STATUS sul *domoDevice* con *web service* http://www.questowebservice.it/servizio e id 1 con input di tipo booleano dal valore TRUE.

Eseguito il servizio, è possibile ricevere il seguente messaggio di conferma:

```
<message message="TRUE" messageType="SUCCESS"
 receiverURL="http://www.questowebservice.it/servizio"
 receiverId="0" senderId="1"
 senderURL="http://www.altrowebservice.it/servizio" />
```

Questo messaggio dà la conferma al mittente che il precedente messaggio è stato eseguito con successo e che la lampada è ora accesa.

Capitolo 6

Lato server: il web service

Il lato server dell'applicazione risolve la cooperazione tra i dispositivi che appartengono a diverse tecnologie e esegue i comandi provenienti da altri server o da un client.

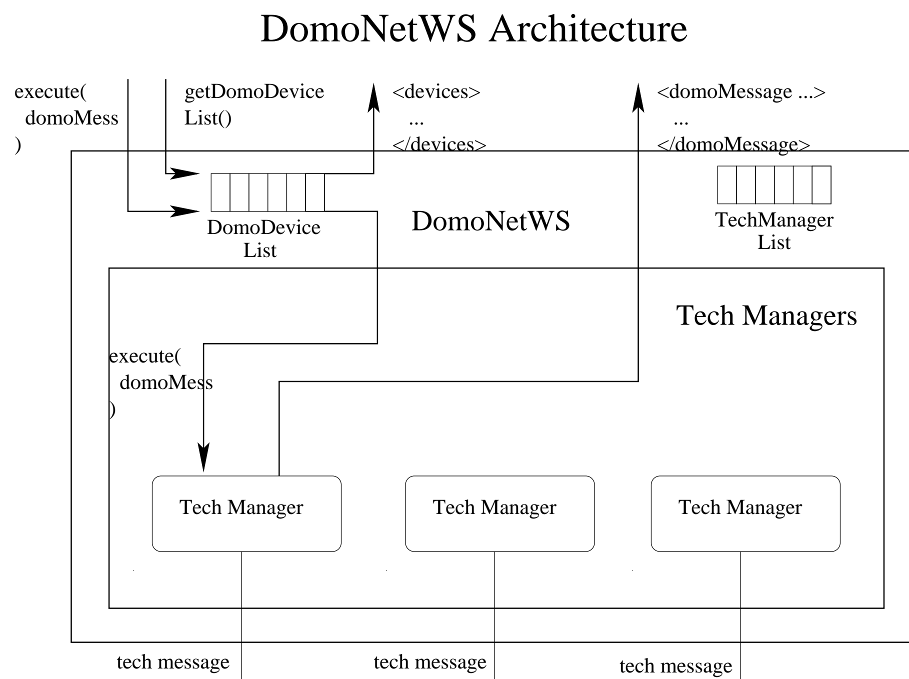

Figura 6.1: L'architettura del domoNetWS (web service).

Il *web service*, al momento in cui viene montato, inizializza i *tech manager* elencati nella `techManagerList` dai quali riceve la lista dei *device* disponibili. In un *web service* ci possono essere più *tech manager* differenti.

Ogni dispositivo riconosciuto da ogni *tech manager* viene convertito usando il formalismo *domoDevice* e memorizzato nella `DomoDeviceList` dove viene indicizzato assegnandogli un *domoAddress* composto dalla *url* del *web service* di appartenenza, e da un *id* (un numero progressivo intero che permette di identificarlo all'interno).

6.1 Eseguire un *domoMessage*

La richiesta di eseguire un *domoMessage* può venire da un'applicazione client (per il controllo a distanza dei dispositivi), dallo stesso o da un altro *web service* (per la cooperazione).

Quando arriva un *domoMessage* per essere eseguito, per prima cosa viene individuato il *domoDevice* del dispositivo interessato grazie al `DomoDeviceId` calcolato usando gli attributi `receiverURL` e `receiverId` del messaggio. Dal *domoDevice* viene individuato il tipo di tecnologia di appartenenza del dispositivo e, con la `techManagerList`, individuato il *tech mamanger* di competenza al quale forwardare il messaggio. Il *tech manager* traduce il *domoMessage* nel suo corrispettivo *tech message* e provvede alla sua esecuzione. Eseguito il messaggio, il *tech manager* provvede a costruire un nuovo *domoMessage* di suc-

cesso (`type="SUCCESS"`) o fallimento (`type="FAILURE"`) con un eventuale risposta da parte del dispositivo.

6.2 La cooperazione tra differenti *tech manager*

A questo livello, la cooperazione tra dispositivi appartenenti a diverse tecnologie funziona come mostrato (figura 6.2):

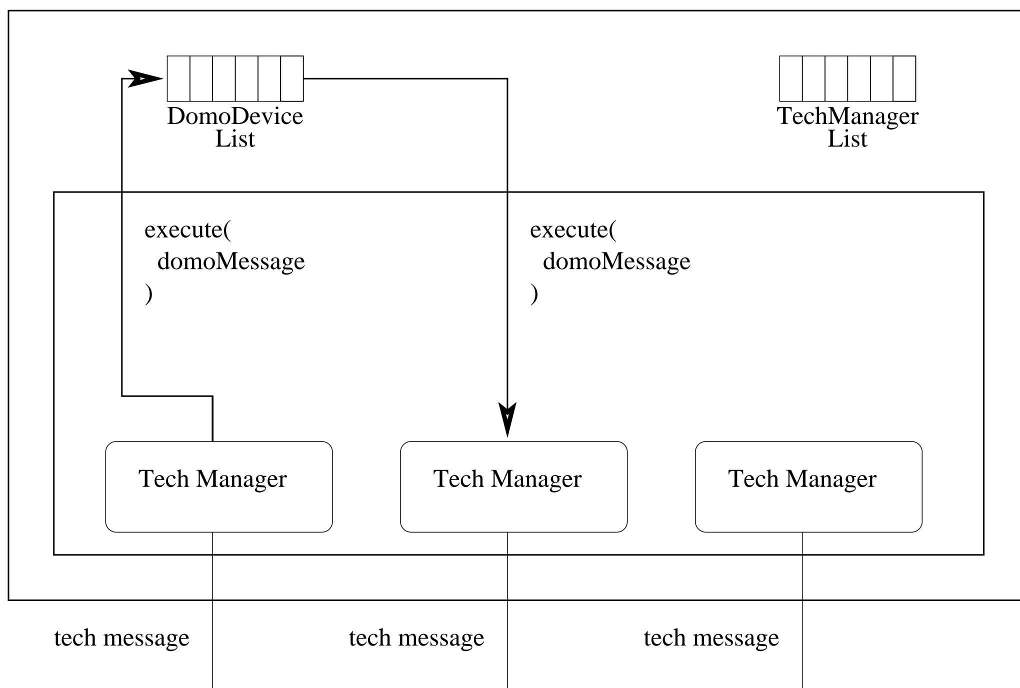

Figura 6.2: L'architettura del domoNetWS (web service): la cooperazione.

Il *tech manager* riceve un *tech message* dalla propria rete ed individua, analizzandolo, il *domoDevice* rappresentante il mittente ed il servizio coinvolto. I dati trovati vengono ana-

lizzati dal *web service* che provvede a verificare l'esistenza di `linkedService` nella descrizione del servizio invocato. Se trovati, genera i nuovi *domoMessage* e li invia ai *web service* interessati altrimenti non compie alcuna azione.

Ogni *web service* infatti può anche importare e gestire i dispositivi appartenenti ad altri *web service* sfruttando la rete Internet e comunicando direttamente con il *domoNetWS* interessato. All'interno della `domoDeviceList` i *domoDevice* importati sono riconosciuti attraverso l'attributo `url`. In questo modo è possibile avere l'interoperabilità tra dispositivi domotici appartenenti a *web service* diversi (figura 6.3).

6.3 Il *tech manager*

Il *tech manager* è un modulo (o driver) del *web service* che ha la funzione di interfacciarsi fisicamente con i dispositivi appartenenti ad una determinata tecnologia come *Konnex*, *UPnP* etc. Il *tech manager* implementa le chiamate, funzioni e tutto quanto necessario alla corretta interazione con i dispositivi.

Le funzioni principali a cui deve essere in grado di adempiere sono:

- la generazione di una lista di *domoDevice* disponibili;
- tradurre un *domoAddress* in un *real address* e vice versa;
- tradurre un *domoMessage* in un *tech message* e vice versa.

DomoNetWS Architecture

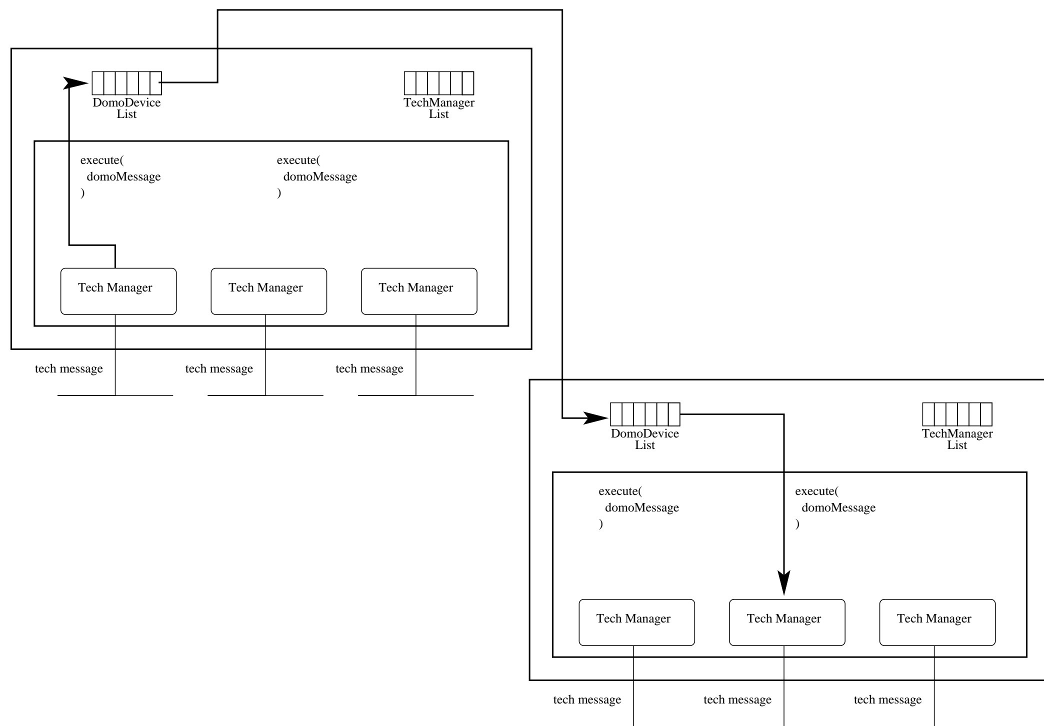

Figura 6.3: L'architettura del domoNetWS (web service): la cooperazione tra diversi web service*.*

In fase di inizializzazione, il *tech manager* deve avere la lista dei dispositivi disponibili ed inizializzare eventuali strutture dati.

6.3.1 Esecuzione di un *domoMessage*

Quando il *techManager* riceve un *domoMessage*, esso ricava da quest'ultimo il *real address* del dispositivo destinatario del messaggio e, usando i dati del messaggio, genera il corrispettivo *tech message*.

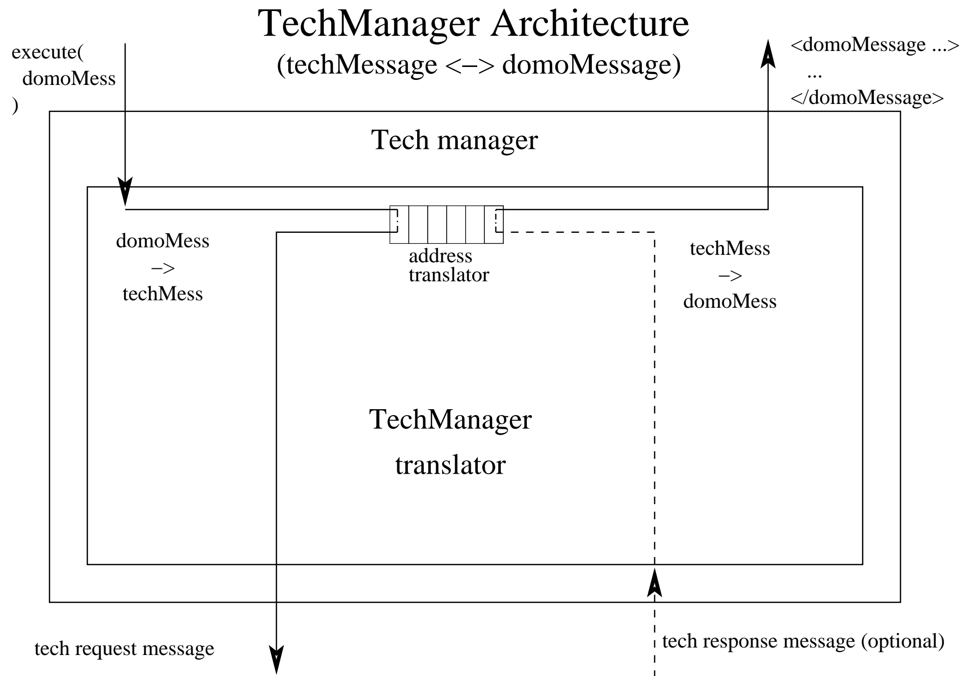

Figura 6.4: L'architettura del techManager: l'esecuzione di un domoMessage.

Questo significa che il *tech manager* deve verificare se il *domoMessage* è di tipo `COMMAND`, e capire l'azione da eseguire interpretando l'attributo `message`. Deve convertire i tag `input` (se presenti) usando gli attributi `datatype` e `value` in modo da poter tradurre correttamente i valori da propagare. A questo punto il messaggio (ora *tech message*) può essere inviato usando il mezzo di comunicazione previsto per la specifica tecnologia in uso.

L'esecuzione di un *tech message* può (e non deve) coinvolgere altri messaggi come ad esempio una risposta al messaggio con un valore o con un *ack*. Se è prevista una risposta, deve essere convertita da *tech message* in *domoMessage* considerando ancora i

datatype ed i valori. Queste informazioni vengono memorizzate all'interno del *domoMessage* di tipo `SUCCESS` o `FAILURE`. Se non è prevista una risposta, il *tech manager* restituisce un *domoMessage* vuoto comunque sempre di tipo `SUCCESS` o `FAILURE` in dipendenza dallo stato della rete, dal rilevamento di eccezioni, etc.

Per fare tutto ciò, il *tech manager* deve avere al suo interno un traduttore da *tech message* a *domoMessage* e vice versa.

Per tradurre gli indirizzi, il *tech manager* usa una `DoubleHash`, una *hashMap* speciale che permette di usare il *domoDeviceId* o il *real address* come chiave per ottenere l'altro. Per ogni *domoDeviceId* è possibile associare più di un *real address* e per ogni *real address* è possibile associare un solo *domoDeviceId*.

Capitolo 7

Lato client: il domoNetClient

Il *DomoNetClient* esegue il controllo dei dispositivi in remoto indipendentemente dalla loro tecnologia.

Attua la connessione ad uno o più *web service* contemporaneamente prendendo la lista dei *domoDevice* attraverso il metodo `getDeviceList()`, e invia comandi *domoMessage* attraverso il metodo `execute(dm)`. Per qualche dispositivo è previsto anche un parametro di ritorno.

Tutti i *domoDevice* sono memorizzati nella `domoDeviceList`.

Il *domoNetClient* gestisce solamente la parte logica del lato client. Per poter interagie con essa, è necessario implementare una *user interface* tramite un'applicazione *web*, testuale o grafica.

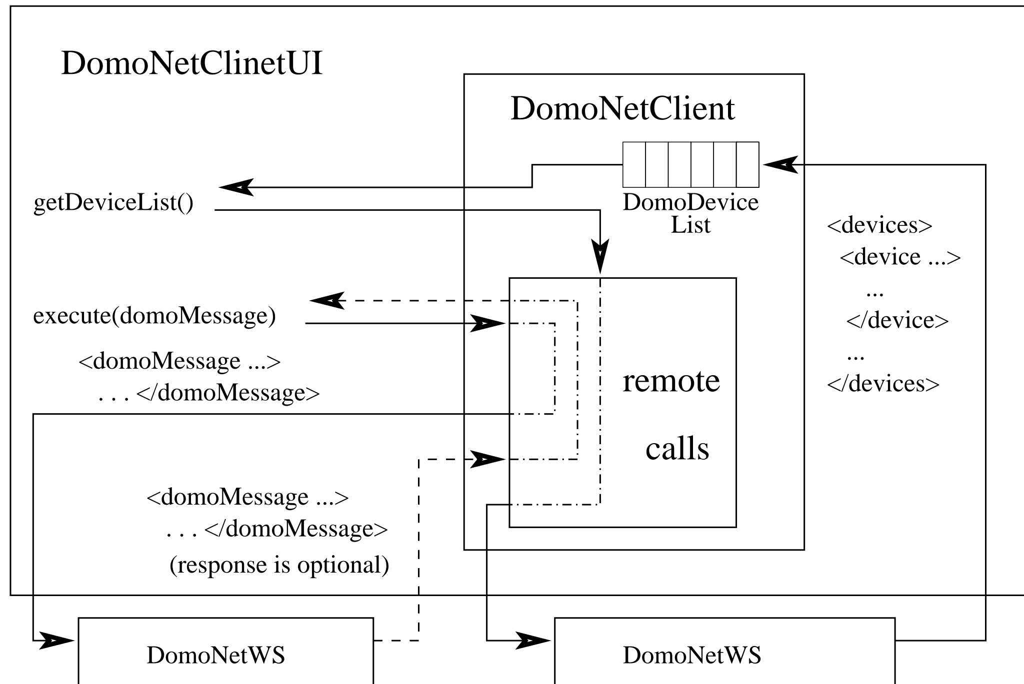

Figura 7.1: L'architettura del domoNetClient.

7.1 Interazione tra il *domoNetClient* ed il *domoNetClientUI*

In figura 7.1 sono mostrate le due possibili interazioni tra la parte logica e l'interfaccia utente:

- al momento della connessione ad un *web service*:
  la *user interface* richiede la lista dei dispositivi disponibili invocando il metodo `List getDeviceList()` sul *domoNetClient*. Il *domoNetClient* esegue una chiamata remota indirizzata verso il *web service* richiesto creando una copia locale dei *domoDevice* nella `DomoDeviceList` indicizzabile

tramite il *domoDeviceId*. Sia il *domoNetClient* che il *domoNetClientUI* condividono questa struttura in sola lettura; Ottenuta la lista, il *domoNetClientUI* provvede a mostrarla all'utente.

- per l'esecuzione di un *domoMessage*:
  la *user interface* richiede l'esecuzione di un servizio generando il relativo *domoMessage* ed invocando il metodo `DomoMessage execute(DomoMessage domoMessage)` sul *domoNetClient*. Il *domoNetClient* esegue una chiamata remota indirizzata verso il *web service* coinvolto e rimane in attesa di una risposta. Ottenuta la risposta, questa viene passata alla *domoNetClientUI* che provvede a mostrarla all'utente.

Capitolo 8

Il prototipo

8.1 Strumenti di lavoro

Il prototipo è stato realizzato grazie alla mia collaborazione con il Laboratorio di Domotica presso l'Istituto di Scienza e Tecnologie dell'Informazione ISTI "Alessandro Faedo" del CNR di Pisa. Il laboratorio mi ha messo a disposizione i dispositivi domotici sui quali è stato effettuato lo studio, le prove ed i test finali.

In particolare al laboratorio ho lavorato con il *middleware Konnex* così costituito (figura 8.1):

- bus Konnex/EIB su TP1;
- BCU (interfaccia tra PC e bus via porta seriale RS232);
- BCU (interfaccia tra PC e bus via TCP/IP);
- attuatore Siemens a 4 uscite, a cui sono state collegate altrettante lampadine;
- dimmer ABB;

- termostato Theben RAM713;
- altro termostato;
- interruttore/regolatore;
- interruttore doppio + 1SA;
- alimentatore.

Figura 8.1: Pannello Konnex.

Inoltre si sono dovuti utilizzare strumenti software che consentissero la configurazione del sistema. Per la configurazione della rete si è utilizzato il software *ETS3* della famiglia *ETS*.

Per *UPnP* non è stato necessario l'acquisto di componenti o di dispositivi fisici per creare l'infrastruttura di rete. Poiché questo *middleware* si basa sui protocolli dello *stack TCP/IP*, è stato sufficiente sviluppare il software su un PC provvisto di una tradizionale interfaccia di rete *Ethernet*. Per la fase di studio, prove e test finali è stato usato il simulatore di dispositivi domotici *UPnP* incluso come esempio nelle librerie *Cyberlink*. In particolare il simulatore mi ha permesso di lavorare con i seguenti dispositivi (figura 8.2):

- dispositivo di aria condizionata;
- orologio digitale con data;
- lampadina;
- televisione;
- lavatrice;
- dispositivo per il controllo remoto (telecomando).

8.2 Lo sviluppo

Il prototipo è un software *open source* rilasciato sotto licenza *GNU*.

Il software è stato scritto in *Java* ed usa esclusivamente strumenti *open source*:

Figura 8.2: I dispositivi domotici UPnP *simulati utilizzati.*

- il *web server Tomcat*[1];
- il contenitore di *web service axis*[2];
- la libreria per la gestione dell'*XML Xerces*[3];
- le librerie per l'interazione con i dispositivi domotici *Calimero*[4] e *Cyberlink*[5].

Per promuovere a pieno titolo lo sviluppo del software *open source* è stato iniziato lo studio di piattaforme *Java* alternative

[1]http://www.apache.org
[2]http://ws.apache.org/axis
[3]http://xerces.apache.org/xerces2-j
[4]http://calimero.sourceforge.net
[5]http//sourceforge.net/projects/cgupnpjava

a quella *Sun*[6] ed altrettanto valide come l'*ecj*: il compilatore *Java open source* di *Eclipse*[7].

Di seguito c'è la descrizione dei *package* sviluppati rimandando per eventuali approfondimenti ai *javadoc* autogenerati dai sorgenti.

8.2.1 Il *domoML*

DomoML è un un linguaggio contenuto in una libreria *XML* basata su *Xerces*[8].

Struttura del *package domoML*

La struttura del *package domoML* che implementa la libreria è:

- `DomoMLDocument`:
  classe che implementa l'interfaccia `org.w3c.dom.Document`. Ereditandola, permette di creare un nuovo *Document XML* compatibile con tutte le funzionalità di *domoML*. La classe, oltre ai metodi previsti dall'interfaccia, implementa:
  - `DataType`:
    enumerato contenente i *datatype* previsti per *domoML*;
  - `Element createElement(String tagName)`:
    metodo per aggiungere al *document* un elemento com-

[6] http://java.sun.com
[7] http://www.eclipse.org
[8] http://xerces.apache.org/xerces2-j

patibile *domoML* e con tag `tagName`. Restituisce l'*Element* creato;

- `java.lang.String toString()`:
  restituisce una *String* contenente il codice *XML* rappresentante il documento.

- `DomoMLElement`:
  classe che implementa l'interfaccia `org.w3c.dom.Element`. Rappresenta un elemento appartenente ad un *DomoMLDocument*.
  Ogni `createElement()` crea un oggetto che eredita da questa classe.

- `domoDevice`:
  contiene le classi che descrivono il *domoDevice*:

  - `DomoDevice`:
    classe che eredita da `DomoMLDocument`. Rappresenta il *document* della descrizione astratta dei dispositivi fatte secondo *domoML*. I metodi e i campi principali implementati per la classe sono:

    * `DomoTech`:
      enumerazione delle tecnologie supportate dai *domoDevice*;

    * `DomoDeviceService addService(...)`:
      aggiunge un servizio al *DomoDevice*;

  * `DomoDeviceService getService(String serviceName)`:
    restituisce il servizio del *domoDevice* con nome `serviceName`;
  * `java.util.List getServices()`:
    restituisce la lista di tutti i servizi del *domoDevice*;

- `DomoDeviceId`:
  permette di identificare univocamente, tramite l'`url` e l'`id`, un *domoDevice*. La classe viene usata per indicizzare principalmente *HashMap*;
- `DomoDeviceIdWithDescription`:
  come il `DomoDeviceId` ma con in più un attributo contenente anche la descrizione del dispositivo;
- `DomoDeviceSAXParser`:
  parser per ottenere da uno *strem XML* una lista o un solo *domoDevice*;
- `DomoDeviceService`:
  classe che eredita da `DomoMLElement`. Descrive il servizio appartenente ad un *domoDevice*. I metodi e i campi principali implementati per la classe sono:
  * `DomoDeviceServiceInput addInput(...)`:
    aggiunge un campo input al servizio;
  * `java.util.List getInputs()`:
    restituisce la lista di tutti gli input legati al servizio;

    * `DomoDeviceServiceInput addLinkedService(...)`:
      aggiunge un campo per implementare la interoperabilità tra *domoDevice*;
    * `java.util.List getLinkedSerivices()`:
      restituisce la lista di tutti i servizi legati ad altri *domoDevice*;

- `DomoDeviceServiceInput`:
  classe che eredita da `DomoMLElement`. Descrive il tipo di input legato ad un servizio appartenente ad un *domoDevice*; I metodi e i campi principali implementati per la classe sono:
    * `DomoDeviceServiceInputAllowed addAllowed(...)`:
      aggiunge un valore ammesso per il tipo di *input*;
    * `java.util.List getAllowed()`:
      restituisce la lista i valori ammessi per l'*input*;
    * `DomoDeviceServiceInput addLinkedService(...)`:
      aggiunge un campo per implementare la interoperabilità tra *domoDevice*;
    * `java.util.List getLinkedSerivices()`:
      restituisce la lista di tutti i servizi legati ad altri *domoDevice*;
- `DomoDeviceServiceInputAllowed`:

classe che eredita da `DomoMLElement`. Descrive un input ammesso per il servizio;

- `DomoDeviceServiceLinkedService`:
  classe che eredita da `DomoMLElement`. Descrive come operare per eseguire l'interoperabilità tra *domoDevice*; I metodi e i campi principali implementati per la classe sono:
  * `DomoDeviceServiceLinkedServiceInput addLinkedServiceInput(...)`:
    aggiunge un'associazione tra gli *input* dei servizi in gioco;
  * `java.util.List getInputs()`:
    restituisce la lista delle associazioni degli *input*;
- `DomoDeviceServiceLinkedServiceInput`:
  classe che eredita da `DomoMLElement`. Descrive l'associazione degli *input* tra i servizi in gioco;

• `domoMessage`:
contiene le classi che descrivono il *domoMessage*:

- `DomoMessage`:
  classe che eredita da `DomoMLDocument`. Rappresenta il *document* della descrizione interazioni tra i dispositivi fatte secondo *domoML*. I metodi e i campi principali implementati per la classe sono:

    * `MessageType`:
      enumerazione dei tipi di messaggi ammessi;
    * `DomoMessageInput addInput(...)`:
      aggiunge un valore da allegare al contenuto del *domoMessage*;
    * `DomoMessageInput getInput(String inputName)`:
      restituisce l'*input* del *domoMessage* con nome `serviceName`;
    * `java.util.List getInputs()`:
      restituisce la lista di tutti gli *input* del *domoMessage*;
  - `DomoMessageInput`:
    classe che eredita da `DomoMLElement`. Descrive il tipo di input legato ad un servizio appartenente ad un *domoMessage*;
  - `DomoMessageSAXParser`:
    parser per ottenere da uno *strem XML* un *domoMessage*.

Costruire un *domoDevice*

È possibile costruire un nuovo *domoDevice* invocando i costruttori della classe in due maniere differenti:

1. attraverso righe di codice:

```
// crea un tag chiamato "device"
DomoDevice domoDevice = new DomoDevice();
// riempie il tag
domoDevice.setDescription(
                "lampada a risparmio energetico");
domoDevice.set...
...
```

oppure in una riga:

```
DomoDevice domoDevice =
           new DomoDevice(DomoDevice.DomoTech.KNX,
                                   "lampada", ...);
```

per poi continuare con:

```
// aggiungo il servizio per settare lo status
DomoDeviceService service = domoDevice.addService(
            "SET_STATUS", "Set the status",
            "status");
// e l'unico input che ha
DomoDeviceServiceInput input =
            service.addInput("status", "The value",
                      DomoDevice.DataType.BOOLEAN);
// con i possibili valori
input.addAllowed("TRUE");
input.addAllowed("FALSE");
...
```

2. invocare il costruttore passando una stringa contenente il codice xml:

```
DomoDevice domoDevice =
      new DomoDevice("<device description=\"lampada
          a risparimo energetco\" id=\"0\",
          manufacturer=\"pholips\",
          postionDescription=\"comodino accanto al
          letto\", serialNumber=\"xxxxxxxxx\"
          tech=\"KNX\" type=\"lampada\" url=\"\">
    <service description=\"Get the status\"
     name=\"GET_STATUS\" output=\"BOOLEAN\"
     outputDescription=\"The value\"
     prettyName=\"Get status\" />
    <service description=\"Set the status\"
     name=\"SET_STATUS\"
     prettyName=\"Set status\">
        <input description=\"The value\"
         name=\"status\"
         type=\"BOOLEAN\">
            <allowed value=\"TRUE\" />
            <allowed value=\"FALSE\" />
        </input>
    </service>
</device>");
```

Costruire un *domoMessage*

È possibile costruire un nuovo *domoMessage* invocando i costruttori della classe in due maniere differenti:

1. attraverso righe di codice:

   ```
   // crea un tag chiamato "device"
   DomoMessage domoMessage = new DomoMessage();
   // riempie il tag
   domoMessage.setMessage("SET_STATUS");
   domoMessage.setMessageType(
               DomoDevice.MessageType.COMMAND);
   domoMessage.set...
   ...
   ```

   oppure in una riga:

   ```
   DomoMessage domoMessage = new DomoMessage(
            "http://www.altrowebservice.it/servizio",
            "1",
            "http://www.questowebservice.it/servizio",
            "0",
            "SET_STATUS",
            DomoDevice.MessageType.COMMAND);
   ```

   per poi continuare con:

   ```
   // aggiungo l'input
   ```

```
domoMessage.addInput("status","TRUE",
         DomoMessage.DataType.BOOLEAN);
```

2. invocare il costruttore passando una stringa contenente il codice xml:

```
DomoMessage domoMessage =
  new DomoMessage("<message message=\"SET_STATUS\"
    messageType=\"COMMAND\"
    receiverId=\"1\"
    receiverURL=
      \"http://www.altrowebservice.it/servizio\"
    senderId=\"0\"
    senderURL=
      \"http://www.questowebservice.it/servizio\">
    <input name=\"status\" type=\"BOOLEAN\"
     value=\"TRUE\" />
</message>");
```

8.2.2 Il *web service*

Il *web service* è implementato usando il *web container Tomcat* ed il contenitore di servizi *axis*, entrambi *open source*.

Struttura del *package domoNetWS*

La struttura del *package domoNetWS* che implementa il *web service* è:

- `DomoNetWS`:
  classe che implementa un *web service*. I metodi principali implementati sono:
  - `deviceList`:
    HashMap che contiene la lista dei *domoDevice* indicizzati per `DomoDeviceId`;
  - `managerList`:
    HashMap che contiene la lista dei *tech manager* indicizzati per `domoML.domoDevice.DomoDevice.DomoTech`;
  - `DomoDeviceId addDomoDevice(DomoDevice domoDevice)`:
    aggiunge un *domoDevice* alla lista dei *domoDevice* assegnando un `DomoDeviceId` univoco;
  - `java.lang.String execute(String messageString)`:
    esegue un *domoMessage* riconoscendo il *domoDevice* destinatario del messaggio;
  - `void finalize()`: azioni da eseguire al termine allo *shutdown* del *web service*;
  - `DomoDevice getDomoDevice(DomoDeviceId domoDeviceId)`:
    restituisce un *DomoDevice* dandone il descrittore;
  - `java.lang.String getDomoDeviceList()`:

restituisce uno *stream XML* rappresentante la lista dei *domoDevice* raggiungibili dal *web service*;

  - `void printLicenceNote()`:
    mostra la licenza del *software*;
  - `boolean removeDomoDevice()`:
    rimuove un *domoDevice* dalla lista dei *domoDevice* disponibili;
  - `void searchAndExecuteLinkedServices(...)`:
    ricerca eventuali *tag* per l'interoperazione tra i *domoDevice*.

- `techManager`:
  contiene le classi che implementano i *tech manager*.
  - `DoubleHash`:
    classe che implementa una *HashMap* bidirezionale (può essere indizzato sia tramite un *real address* per ottenere un *domoDeviceId* e vice versa.);
  - `TechManager`:
    classe con metodi astratti dalla quale devono ereditare tutti i *tech manager* in modo che siano compatibili con *domoNet*. I metodi astratti previsti sono:
    * `void addDevice(DomoDevice domoDevice,`
      `String address)`:
      aggiunge un *domoDevice* al *tech manager* fornendo il *real address*;

  * `DomoMessage execute(DomoMessage domoMessage)`:
    esegue un *domoMessage* e restituisce una risposta al termine;
  * `void finalize()`:
    azioni da compiere alla chiusura del *tech manager*;

- `knxManager`:
  classe che eredita da `TechManager`. contiene le classi che implementano le funzionalità per la tecnologia *Konnex*.
  * `KNXManager`:
    implementa le funzionalità per interagire con i dispositivi *Konnex*;
  * `KNXManagerConfigurationSAXParser`:
    classe per parsare il file di configurazione di *ETS*;

- `upnpManager`:
  classe che eredita da `TechManager`. contiene le classi che implementano le funzionalità per la tecnologia *UPnP*.
  * `UPNPManager`:
    implementa le funzionalità per interagire con i dispositivi *UPnP* e col *web service*;
  * `UPNPManagerPoint`:
    listener per gestire i dispositivi *UPnP*;

Come implementare un nuovo *tech manager*

Per implementare un nuovo modulo *tech manager*, è necessario estendere la classe `domoNetWS.techManager.TechManager` e creare una nuova istanza della nuova classe nella `domoNetWS.DomoNetWS.managerList` usando come chiave l'identificatore della tecnologia (essa deve essere contenuta in `domoML.domoDevice.DomoDevice.DomoTech`). Il nuovo modulo deve implementare i metodi astratti che sono:

- `public void addDevice(final DomoDevice domoDevice, String address)`:
  per aggiungere un dispositivo alla lista dei *domoDevice* del *web service*; deve chiamare i metodi `doubleHash.add()` e `addDomoDevice()` del *web service*;

- `public DomoMessage execute(DomoMessage domoMessage) throws Exception`:
  per eseguire un `domoML.domoMessage.DomoMessage`; deve essere in grado di convertire il *domoMessage* in *tech message* e vice versa;

- `public void finalize()`:
  azioni da eseguire quando viene chiuso il *web service*.

I *tech manager* implementati

Questi sono i *tech manager* implementati e che possono essere presi come esempio per quelli nuovi.
KNXManager
Il *KNXManager* è il modulo che gestisce la tecnologia *Konnex* e si basa sulle librerie *open source Calimero*[9].

Il costruttore del modulo prende come parametro la *url* e il numero di porta del servizio che è connesso ai dispositivi.

Al tempo di *startup* esegue la connessione al server e l'inizializzazione delle strutture che si occupano della conversione dei *datatype*, esattamente da `domoML.DomoDevice.domoDevice.DataType` agli identificatori `Major` e `Minor`, usati per convertire il *tag* `input` del *domoMessage* in un valore fruibile per la tecnologia. Dopo le inizializzazioni il modulo carica i dispositivi disponibili parsando un file *XML* generato dall'applicazione *ETS 3*[10] già presentata nel capitolo riguardante *Konnex*.

Il file di configurazione contiene le descrizioni e tutte le funzionalità che il sistema *Konnex* può offrire.

Come esempio, di seguito è mostrato un pezzo di file di configurazione per il servizio di un dispositivo:

```
<row>
  <colValue nr="1">0/0/1 Comando Uscita A</colValue>
```

[9] http://calimero.sourceforge.net
[10] *ETS* (EIB Tools Software) è un software distribuito dalla *Konnex Association* per settare e configurare dispositivi.

```
  <colValue nr="2">
    0: Comando,tasto sinistro superiore-Commutazione
  </colValue>
  <colValue nr="3">
    1.1.4 16196.. Pulsante 4 canali
  </colValue>
  <colValue nr="4">S</colValue>
  <colValue nr="5">-</colValue>
  <colValue nr="6">C</colValue>
  <colValue nr="7">-</colValue>
  <colValue nr="8">W</colValue>
  <colValue nr="9">T</colValue>
  <colValue nr="10">U</colValue>
  <colValue nr="11">
    16196.. Pulsante 4 canali
  </colValue>
  <colValue nr="12">
    16196 On-Off-Dimmer-Tapparelle-Led
  </colValue>
  <colValue nr="13">1 bit</colValue>
  <colValue nr="14">Basso</colValue>
  <colValue nr="15">0/0/1</colValue>
</row>
```

Da questo pezzo, e da ogni tag `row` presente nel file, è possibile individuare:

- nel secondo `colValue`: il nome del servizio;
- nel terzo `colValue` (la parte iniziale): il *real address* del dispositivo;
- nel settimo `colValue`: se il campo è leggibile (settato a `R`);
- nell'ottavo `colValue`: se il campo è scrivibile (settato a `W`);
- nel nono `colValue`: se il campo è trasmissibile (settato a `T`);
- nell'undicesimo `colValue`: il nome del device che offre il servizio;
- nel dodicesimo `colValue`: la descrizione del servizio;
- nel quindicesimo `colValue`: l'indirizzo di gruppo associato al servizio;

Gli altri `colValue` non sono significativi allo scopo proposto dal *KNXManager*. Interessano i servizi che sono trasmissibili e almeno o leggibili o scrivibili.

Un dispositivo viene descritto in tutti i suoi servizi unendo insieme tutti gli undicesimi `colValue` con la stessa etichetta.

Questo esempio produce il seguente *domoDevice*:

```
<device description="" id="" manufacturer=""
 postionDescription="" serialNumber="" tech="KNX"
 type="16196.. Pulsante 4 canali" url="">
    <service description="Get the status"
```

```
  output="BOOLEAN" outputDescription="The value"
  prettyName="Get status" name="GET_STATUS" />
 <service name="0/0/1"
  description="16196 On-Off-Dimmer-Tapparelle-Led"
  prettyName="16196 On-Off-Dimmer-Tapparelle-Led">
    <input description="The value" name="value"
     type="BOOLEAN">
       <allowed value="TRUE" />
       <allowed value="FALSE" />
    </input>
 </service>
</device>
```

Come catturare i tech message *dal bus* Konnex

Tutti i messaggi che transitano sul bus *Konnex* vengono catturati dal *KNXManager* attraverso l'uso di un *listener* in ascolto sul *bus*. È interessante monitorare il *bus* per due scopi:

1. in attesa di risposta derivata dall'esecuzione di un *domoMessage*:
   in questo caso è stato implementato un sistema a semafori dove i produttori sono i dispositivi che scrivono sul bus e il *tech manager* è il consumatore. Il *real address* del mittente di ogni messaggio che transita sul bus viene confrontato con quello di cui siamo in attesa. Se l'indirizzo viene riconosciuto in un limite di tempo prefissato viene generato con i dati del *tech message* il *domoMessage* di risposta al-

trimenti viene generato un *domoMessage* di fallimento. Un esempio tipico di questo meccanismo è l'interrogazione di un dispositivo per conoscerne lo stato;

2. per implementare la cooperazione:
   ogni messaggio che transita sul bus viene catturato dal *listener*. Il *tech message* estrapola il *real address* del dispositivo mittente e riconosce il servizio tramite l'indirizzo di gruppo a cui è rivolto il comando. Con questi dati, è possibile ricavare il *domoDevice* e la descrizione del servizio. Il *web service* può quindi verificare se è impostato il tag `linkedService` ed agire di conseguenza.

UPnPManager L'*UPnPManager* è il modulo che gestiche la tecnologia *UPnP* e si basa sulla libreria *Cyberlink*[11].

A tempo di *startup*, il l'*UPnPManager* inizializza il `ManagerPoint`: il *listener*, cuore del manager, che permette di aggiungere, rimuovere, testare l'*hartbeat* e amministrare i pacchetti dei dispositivi. Il `ManagerPoint` provvede ad inizializzare le strutture dati per la conversione da
`domoML.DomoDevice.domoDevice.DataType` ai simboli usati per questa tecnologia e vice versa.

Quando viene aggiunto un dispositivo alla rete *UPnP*, questo si annuncia dando la propria descrizione e quella dei servizi che può offrire. Queste informazini sono memorizzate in diversi file *XML* che la libreria provvede a parsare e a darne l'albero

[11] http//sourceforge.net/projects/cgupnpjava

corrispondente. A questo punto diventa facile ottenere le informazioni cercate. Un esempio di descrizione del dispositivo è:

```
<device>
 <deviceType>urn:schemas-upnp-org:device:light:1
 </deviceType>
 <friendlyName>CyberGarage Light Device</friendlyName>
 <manufacturer>CyberGarage</manufacturer>
 <manufacturerURL>http://www.cybergarage.org
 </manufacturerURL>
 <modelDescription>CyberUPnP Light Device
 </modelDescription>
 <modelName>Light</modelName>
 <modelNumber>1.0</modelNumber>
 <modelURL>http://www.cybergarage.org</modelURL>
 <serialNumber>1234567890</serialNumber>
 <UDN>uuid:cybergarageLightDevice</UDN>
 <UPC>123456789012</UPC>
 <iconList>
  ...
 </iconList>
 <serviceList>
  ...
 </serviceList>
 <presentationURL>http://www.cybergarage.org
```

```
 </presentationURL>
</device>
```

Da questo pezzo di codice si trova:

- dal tag `friendlyName`: la descrizione del dispositivo;
- dal tag `deviceType`: il *real address* del dispositivo;
- dal tag `modelName`: il tipo del dispositivo;
- dal tag `serialNumber`: il serial number.

Quello dei servizi offerti (chiamati `actions`) è:

```
<actionList>
 <action>
  <name>SetPower</name>
  <argumentList>
   <argument>
    <name>Power</name>
    <relatedStateVariable>
       Power
    </relatedStateVariable>
    <direction>in</direction>
   </argument>
   <argument>
    <name>Result</name>
    <relatedStateVariable>
```

```
       Result
    </relatedStateVariable>
    <direction>out</direction>
   </argument>
  </argumentList>
 </action>
 <action>
  <name>GetPower</name>
  <argumentList>
   <argument>
    <name>Power</name>
    <relatedStateVariable>Power</relatedStateVariable>
    <direction>out</direction>
   </argument>
  </argumentList>
 </action>
</actionList>

<serviceStateTable>
 <stateVariable sendEvents="yes">
  <name>Power</name>
  <dataType>boolean</dataType>
  <allowedValueList>
   <allowedValue>0</allowedValue>
   <allowedValue>1</allowedValue>
```

```
  </allowedValueList>
  <allowedValueRange>
   <maximum>123</maximum>
   <minimum>19</minimum>
   <step>1</step>
  </allowedValueRange>
 </stateVariable>
 <stateVariable sendEvents="no">
  <name>Result</name>
   <dataType>boolean</dataType>
 </stateVariable>
</serviceStateTable>
```

Da questo pezzo di codice si deduce:

- dal tag `action - name`: il nome e il *pretty name* del servizio;
- dal tag `argument - name`: il nome del parametro di input;
- dal tag `relatedStateVariable`: la chiave da cercare nella `serviceStateTable` per trovare le altre informazioni del parametro;
- dal tag `direction`: se è `out` è un parametro di output quindi è aspettato un valore di ritorno all'esecuzione del servizio; se è `in` è un parametro di input;
- dal tag `datatype`: il datatype del parametro;

- dal tag `allowedValue`: i valori permessi per quel parametro.

Per ogni `action` è previsto un set di `argument`. Ogni `argument` ha un `datatype` da essere poi convertito in `domoML.DomoDevice.domoDevice.DataType`. Di ogni `argument` occorre verificare il tag `direction` per sapere se la variabile di stato deve essere usata come parametro di *input* o di *output*.

Il corrispondente *domoDevice* risultante è:

```
<device description="CyberGarage Light Device" id=""
 manufacturer=""  positionDescription=""
 serialNumber="1234567890" tech="UPNP" type="Light">
   <service description="" name="SetPower"
    output="BOOLEAN" prettyName="SetPower">
     <input description="" name="Power"
      type="BOOLEAN">
       <allowed value="0" />
       <allowed value="1" />
     </input>
   </service>
   <service description="" name="GetPower"
    output="BOOLEAN" prettyName="GetPower" />
</device>
```

Come catturare i tech message *dalla rete* UPnP

L'`UPnPManagerPoint` implementa anche le funzioni di *listener* in

attesa di messaggi che transitano sulla rete. Quando un nuovo messaggio transita, ne viene calcolato il codice di sottoscrizione del servizio e il *domoDeviceId* del dispositivo mittente. Con queste informazioni, passate al *web service*, è possibile verificare la presenza di `linkedService` associati al servizio invocato.

8.2.3 Il *DomoNetClient*

il *DomoNetClient* permette di controllare in remoto i dispositivi domotici dislocati nella rete *DomoNet* all'interno dei *web service.*

Struttura del *package domoNetClient*

La struttura del *package domoNetClient* che implementa il *client* è:

- `DomoNetClient`:
  classe che implementa la parte logica del lato *client.* Per poter essere utilizzata occore utilizzare una classe che implementa una *user interface* grafica, testuale, web etc. I metodi e i campi principali implementati per la classe sono:
  - `domoDeviceList`:
    HashMap rappresentante una copia locale dei *domoDevice* scaricati al momento della connessione ai *web service* interessati;
  - `void connectToWebService(String URL,`
    `String description)`:

si connette ad un *web service* richiedendo la lista dei dispositivi disponibili;

  - `java.lang.String execute(DomoMessage domoMessage)`:
    esegue un *domoMessage* individuando il *web service* interessato.

- domoNetClientUI:
  contiene le classi che descrivono il *domoNetClientUI*, il *client* grafico:

  - `DomoNetClientUI`:
    classe che costruisce il *client* grafico da attaccare al `DomoNetClient`;

  - `DefaultWebServicesDescriptorsSAXParser`:
    parser del file di configurazione del *client* grafico.

Un *domoNetClientUI* grafico

Questa interfaccia implementa un client grafico da attaccare al motore *DomoNetClient*.

Il client è fornito da un file di configurazione basato su *XML* dal quale prende le informazioni sui possibili *web service* disponibili. Gli indirizzi dei *web service* sono mostrati in alto contestualmente ad una breve descrizione. La barra dell'indirizzo può essere comunque editata.

Selezionato o editato l'indirizzo del *web service*, cliccando sul

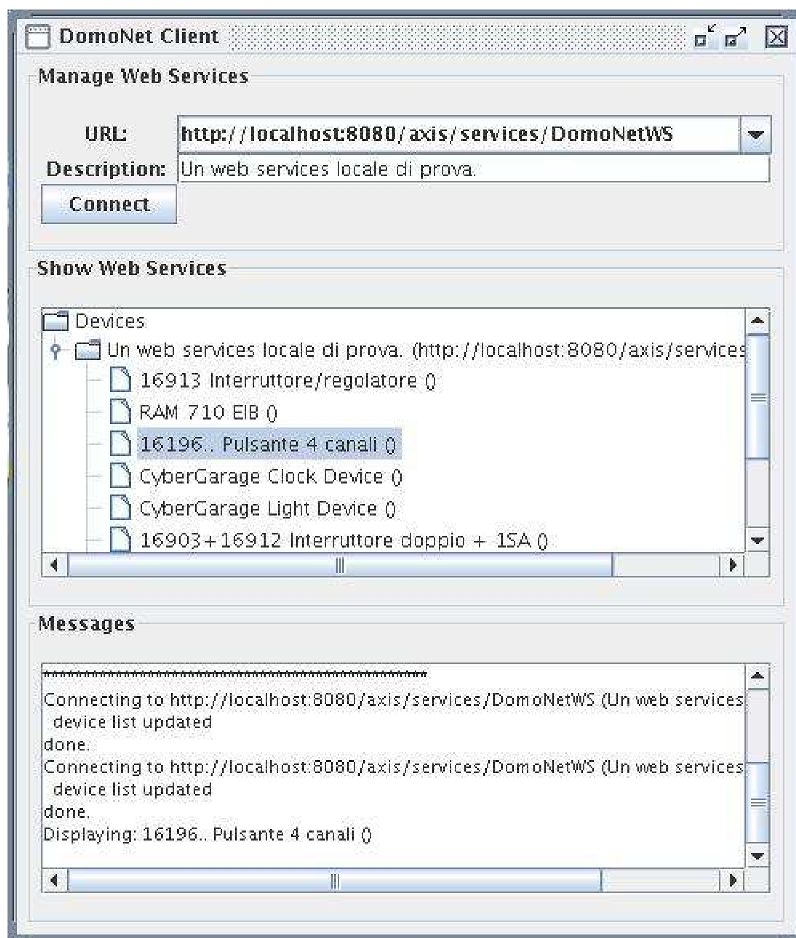

Figura 8.3: Snapshot del domoNetClientUI (finestra principale).

bottone `Connect` si esegue materialmente la connessione al *web service*. Il *web service* risponde dando una lista dei *domoDevice* attualmente disponibili (l'insieme dei dispositivi di tutte le tecnologie supportate).

I *domoDevice* vengono mostrati divisi per *web service* con una struttura ad albero (figura 8.3).

Cliccando su uno di essi viene aperta una nuova finestra costruita a *run time* parsando la descrizione del *domoDevice*. La

finestra mostra delle informazioni generali sul dispositivo ed i servizi disponibili.

Figura 8.4: Snapshot del domoNetClientUI (finestra dei servizi 1).

Per ogni servizio viene mostrato il bottone etichettato con il `pretty name`. Alla sua destra è presente un campo testuale se, all'esecuzione del servizio, è atteso un valore di ritorno. Alla sua sinistra esistono tanti campi testuali quanti parametri di input richiesti per quel servizio. Accanto ad ogni campo testuale è presente una descrizione del tipo di dato atteso.

Riempiendo correttamente i parametri di input richiesti e cliccando sul bottone del servizio, il client genera il *domoMessage* corrispondente da inviare al *domoNetClient* il quale provvede ad instradarlo al *web service* di competenza.

In figura 8.4 è mostrata la finestra dei servizi relativi ad un televisore:

- `SetPower`: per settare lo stato del televisore. É necessario

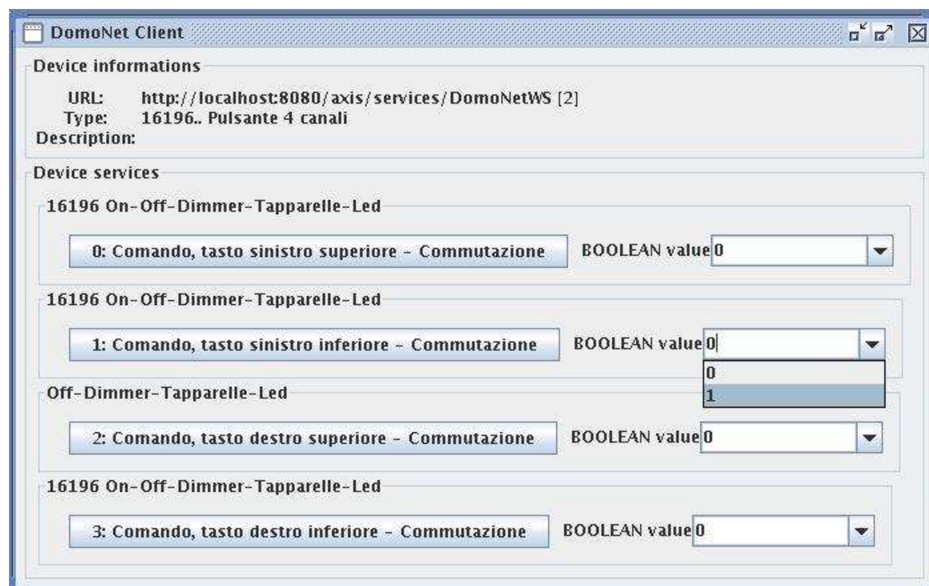

Figura 8.5: Snapshot del domoNetClientUI (finestra dei servizi 2).

un parametro *booleano* in *input* e atteso un parametro di *output* come risposta;

- `GetPower`: restituisce lo stato attuale del televisore con un parametro di *output*.

In figura 8.5 è mostrata la finestra dei servizi di una pulsantiera a quattro canali con un servizio per pulsante. Su di essa è possibile invocare solamente servizi che determinano la commutazione dei pulsanti tramite un valore *booleano* “0” o “1”.

8.3 Il test

Sono stati effettuati numerosi *test* di valutazione per verificare il corretto funzionamento del *software* sia per quanto riguarda

il controllo remoto sia per quanto riguarda l'interoperabilità e non sono stati riscontrati problemi.

Per brevità viene riportato solamente il caso più interessante e semplice tra i *test* effettuati: l'interoperazione tra un interruttore *Konnex* per accendere una lampada *UPnP* e l'interoperazione tra un interruttore *UPnP* per accendere una lampada *Konnex*.

8.3.1 Interruttore *Konnex* con lampada *UPnP*

Come interruttore viene usato l'attuatore *Siemens* a 4 uscite e la lampada *UPnP* simulata. Il rispettivo *domoDevice* dell'attuatore è:


```
<device serialNumber="" tech="KNX"
 id="0" manufacturer="" positionDescription=""
 description="16196 On-Off-Dimmer-Tapparelle-Led"
 type="16196.. Pulsante 4 canali" url="">
   <service
    description="16196 On-Off-Dimmer-Tapparelle-Led"
    name="0/0/1"
    prettyName="0: Comando,tasto sinistro superiore">
      <input description="" name="value"
       type="BOOLEAN">
        <allowed value="0" />
        <allowed value="1" />
      </input>
```

```
</service>
<service
 description="16196 On-Off-Dimmer-Tapparelle-Led"
 name="0/0/2"
 prettyName="1: Comando,tasto sinistro inferiore">
  <input description="" name="value"
   type="BOOLEAN">
    <allowed value="0" />
    <allowed value="1" />
  </input>
  <linkedService id="1" service="SetPower" url="">
    <linkedInput from="value" to="Power" />
  </linkedService>
</service>
<service
 description="16196 On-Off-Dimmer-Tapparelle-Led"
 name="0/0/3"
 prettyName="2: Comando, tasto destro superiore">
  <input description="" name="value"
   type="BOOLEAN">
    <allowed value="0" />
    <allowed value="1" />
  </input>
</service>
<service
```

```
    description="16196 On-Off-Dimmer-Tapparelle-Led"
    name="0/0/4"
    prettyName="3: Comando, tasto destro inferiore">
     <input description="" name="value"
      type="BOOLEAN">
       <allowed value="0" />
       <allowed value="1" />
     </input>
   </service>
</device>
```

Con la pressione del tasto sinistro inferiore dell'attuatore (figura 8.6)viene implementata l'interoperabilità con il *domoDevice* sullo stesso *web service* e con *id* "1".

Il rispettivo *domoDevice* della lampada è:

```
<device description="" id="1" manufacturer=""
 positionDescription="" serialNumber="1234567890"
 tech="UPNP" type="CyberGarage Light Device">
   <service description="" name="SetPower"
    output="BOOLEAN" prettyName="SetPower">
     <input description="" name="Power"
      type="BOOLEAN" />
   </service>
   <service description="" name="GetPower"
    output="BOOLEAN" prettyName="GetPower" />
</device>
```

Figura 8.6: Pulsantiera Konnex *a 4 canali.*

Alla pressione del pulsante la lampada si accende (figura 8.7) e alla ripressione la lampada si spegne.

8.3.2 Interruttore *UPnP* con lampada *Konnex*

Per questo *test* invece di interoperare con lo stato di un interruttore, interoperiamo con lo stato della lampada *UPnP* e con l'attuatore *Konnex* del paragrafo precedente. Il risultato atteso è l'accensione e lo spengimento contemporaneo della lampada *UPnP* e *Konnex*, dimostrando così la flessibilità del software sviluppato.

Il *domoDevice* della lampada *UPnP* è:

Figura 8.7: Accensione lampada UPnP *alla pressione del tasto della pulsantiera* Konnex.

```
<device description="CyberGarage Light Device" id=""
 manufacturer=""  positionDescription=""
 serialNumber="1234567890" tech="UPNP" type="Light">
   <service description="" name="SetPower"
    output="BOOLEAN" prettyName="SetPower">
     <input description="" name="Power" type="BOOLEAN">
       <allowed value="0" />
       <allowed value="1" />
     </input>
     <linkedService id="0" service="0/0/2" url="">
       <linkedInput from="Power" to="value" />
     </linkedService>
   </service>
   <service description="" name="GetPower"
    output="BOOLEAN" prettyName="GetPower" />
</device>
```

Il *domoDevice* della pulsantiera è analoga al paragrafo prece-

dente ad eccezione dell'assenza del tag `linkedService`.

Cliccando sul tasto *Light Power* del telecomando *UPnP* (figura 8.8)

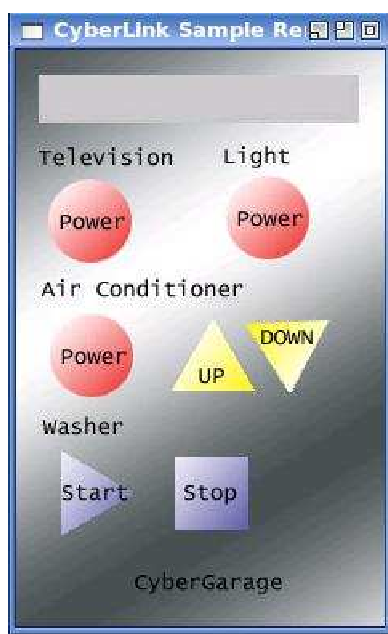

Figura 8.8: Dispositivo di controllo remoto UPnP.

le lampade si accendono (figura 8.9) e alla ripressione le lampade si spengono.

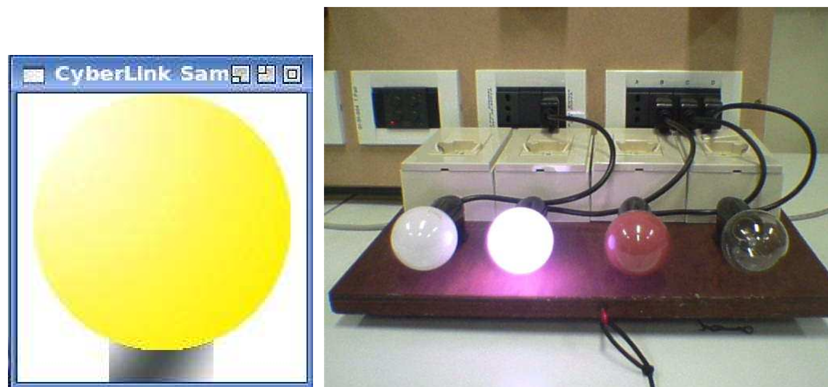

Figura 8.9: Accensione lampade Konnex *e* UPnP.

Capitolo 9

Conclusioni

9.1 Realizzazioni sperimentali e valutazioni

I test sul software effettuati al Laboratorio di Domotica presso l'Istituto di Scienza e Tecnologie dell'Informazione ISTI "Alessandro Faedo" del CNR di Pisa, hanno dimostrato l'efficacia e la generalità del motore del software. Il software è stato in grado di importare correttamente tutti i dispositivi domotici sottoposti alla prova come *dimmer*, termostati, interruttori, televisioni, lavatrici e lampadine. Il lato client è riuscito ad interagire correttamente con i dispositivi sottoposti, mostrando tutti i servizi disponibili, costruendo una corretta interfaccia per ognuno di essi. Per la cooperazione, il punto cruciale è l'individuazione del messaggio che passa sulla via di comunicazione della tecnologia in oggetto, il riconoscimento del dispositivo mittente (per trovare il corrispettivo *domoDevice*) ed il corrispondente servizio invocato. Per quel che riguarda le tecnologie sviluppate, queste

informazioni sono sempre state individuate ed elaborate correttamente quindi è stato sempre possibile verificare la presenza di richieste di cooperazione ed eventualmente soddisfarle.

Da un punto di vista personale, l'esperienza presso il Laboratorio di Domotica ha contribuito a fornirmi nuovi ed interessanti bagagli tecnici in un settore che, nonostante l'auspicabilissima espansione, stenta ancora a decollare e mantiene di fatto un ruolo di nicchia nel campo dell'informatica moderna. Confrontarmi con i continui interrogativi, sia teorici che pratici, che inevitabilmente sorgono in un progetto di questo calibro, ha rappresentato inoltre una sfida con me stesso: dove le conoscenze e gli strumenti già accessibili finiscono, lì si attivano l'intelligenza e la creatività, e ogni piccolo risultato diventa una conquista e un nuovo stimolo. Gli obiettivi prefissati si possono dire complessivamente raggiunti e, malgrado la modestia e l'umiltà non ci portino ad affermare che la nostra soluzione al problema dell'interoperabilità tra i *middleware* domotici sia in assoluto quella giusta, con un pizzico di convinzione credo almeno di aver dato un qualche contributo.

9.2 Integrazione con Internet

Il modello proposto si integra in maniera perfetta con Internet in quanto alla base del suo funzionamento giocano un ruolo fondamentale i protocolli e gli strumenti *standard* e *open* della rete, come *XML* ed i *web service*. Tali strumenti sono utilizzati per le

comunicazioni tra i diversi componenti architetturali e più precisamente tra *domoNetClient* e *domoNetWS* per implementare il controllo remoto (figura 4.3) e tra *domoNetWS* diversi per l'interoperabilità (figura 6.3).

Inoltre, essendo il prototipo rilasciato sotto i termini di licenza *GNU* ed appoggiandosi esclusivamente a *software open source*[1], può essere pubblicato e distribuito sulla rete senza alcun vincolo.

9.3 Direzioni future di ricerca

La domotica, allo stato attuale, è tuttora ostacolata dal tentativo di ciascuna industria di imporre il proprio standard sugli altri. La presenza di un così vasto numero di standard domotici, oramai ampiamente affermati, indica che difficilmente vi sarà la definitiva consacrazione di uno solo di essi.

Il software realizzato rappresenta una soluzione praticabile che possa garantire la definitiva affermazione della domotica in quanto risolve in maniera definitiva, con un'architettura semplice e pulita, l'ostacolo posto.

Occore ancora però molto lavoro. In primo luogo occorre continuare lo sviluppo del software. Questo implementa solamente le funzionalità di base atte a dimostrare l'efficacia dell'architettura presentata ma ancora lontano da poter sfruttare tutte le potenzialità offerte. Non di minore importanza ricopre lo svi-

[1] vedere il capitolo riguardante il prototipo

luppo di altri *tech manager* in modo da poter ampliare il parco delle tecnologie supportate.

Il software presentato, inoltre, rappresenta solo un esempio di applicazione per architettura proposta. Aumentando ulteriormente il grado di generalità dell'architettura è sicuramente possibile trovare ulteriori ambiti d'utilizzo dove è necessaria una cooperazione tra dispositivi e sensori di qualunque tipo di fatto diversi e incompatibili.

Appendice A

Versione *alpha* del manuale del prototipo

Il manuale allegato al prototipo è creato usando il *software open-source texinfo*[1], il formato ufficiale per la documentazione del progetto *GNU*[2].

A.1 Introduction

The slang of the new technology has gone enriching of terms as *home automation*, *building automation*, *smart home* and so on. They are all synonyms diverted by an other neologism, *domotique*, coined in France and composed by the fusion of the two *domus* and *informatique* terms. Such neologism, recovered in Italy with the *domotica* term, represents the application of the

[1]http://www.gnu.org/software/texinfo
[2]http://www.gnu.org

technologies of information and communication to the domestic world, in order to improve from it the comfort and the control.

In this unexplored panorama of development, the industry and the market has played, and they play still, a role of primary importance for the definitive take-off of the *domotic technologies*. On the one hand the deriving financings from the industry have proposed numerous *middleware*, standard always more sophisticated; from the other the presence of too poorly interoperable standards, all promoted by business different coalitions. This has made the domotics an unexploded bomb.

The boom that the world of the research and that of the industry waits is what is not still happened: the real employment of the *domotic technology* is still hindered by the attempt of each industry to impose its standard on the others. The presence of a so vast number of *domotic standards*, by now broadly established, announces that hardly will be the definitive consecration of one of them and it is as much unlikely that all the coalitions gather about to realize an unique *middleware* that represents the standard de facto for the domotics.

My proposal is to realize a practicable solution that is able to guarantee the definitive affirmation of the domotics. My solution consists in the realization of a *framework*, based on the standard technology of the *web service* and on *XML* grammar, said *domoML*, that allows the integration and the interoperability of the offered services from the *middleware* currently introduces

on the market. For obvious reasons, the proposed model represents only a part of the whole *framework*; in particular it will show as the prototype guarantees the interoperability among two the most important, and among diverge, *domotics middleware*: *Konnex* and *UPnP*.

A.1.1 The domotics

The domotics constitutes all the the technologies that are dealt of the automation of the buildings, and in the first place of the integration of the electronic devices, of the appliances, of the systems of communication and control, that they introduce. The purposes that the *domotic applications* establish are:

- increase in the level of comfort: the *domotic applications* must make more pleasant the space in the house facilitating the management above all for the disabled consumers (for example, standard for the audio/video, for the control and the monitoring of the remote appliances, shine, blinds, gates, etc.);
- attainment of a suitable safety level: the *domotic systems* have to guarantee the safety of the consumer to front situations of emergency (for example, techniques of anti intrusion, survey of fires, escapes of gas, etc.);
- search of techniques for the energetic saving: the employment of the domotics has to allow a management more ac-

curate and efficient than the level of the energetic consumptions (for example, advanced tools to get informations that allow a more equitable distribution of the energetic loads).

It's sure that the domotics proposes futuristic sceneries and of the positive impact that its employment would be able have on the quality of our life. For this reason, last years were been the theatre for the development of solutions, promoted above all from private firms, for extended all the benefits derivated from a new, and presumably flourishing market.

Paradoxically, the technologies and the different *domotic standards*, and them poorly interoperable, are so numerous by representing an obstacle to the expansion of the market. From the point of view of the final consumer it's very complicated, and perhaps even little comprehensible, perceive the necessity to acquire domotic instead of traditional devices. In fact the potential beneficents that would divert from the acquisition of this products doesn't succeed to justify an economic greater effort: the introduction of intelligence, understood as ability of calculation, in an any device, involve additional costs always more small. Besides, on account of the scarce interoperability among various standards, the consumer is forced to acquire only the conforming products to a particular system and it could happend in two conditions: the contemporary acquisition of all the present devices in the building, or a technical knowledge that allowed the knowledge of the standard used in order to acqui-

re further conforming devices. Both the conditions are however of difficult realization because in the most of the cases, the domestic environment is a lot of dynamic: the topology and the removal of his elements change frequently and the devices come acquired in different moments.

Try to think, for example, to the appliances: it is at least rare that all that introduce in a residence are acquired contemporarily. To understand better this limitations are of help a classical example of the *domotic applications*: suppose to have a coffeepot that is able to alight through a radio alarm to a pre-established hour. If the radio alarm and the coffeepot speak the same tongue (have the same standard), every morning will be possible have the ready coffee just wakes. Yet, if for any motive, one of the two devices must be changed, will be necessary to replace it with an other in proportion as the same standard.

Is instead desirable allow to the consumer to choose the devices independently from the standard which belong: in this way the consumer don't has not to leastly know the technical detailed bills of the really system, but would be able exclusively draw all the beneficent possibles.

Domotic middlewares

The *domotic middlewares* can be divided in two great classes that differ substantially for the system of communication used:

- bus systems: more traditional and consolidated in the *do-*

motic world. There are standards as X10, EIB, BatiBus, EHS, Konnex, LonWorks, CEBus;

- protocolos more evolved as TCP/IP: more recent as UPnP, Jini, OSGi.

To the side of a particular system of communication, is available a number always greater of standards that realise transmissions as IEEE 802.11b (Wi-penalties), Bluetooth, IEEE 1394 (Firewire), IrDA, ZigBee, etc.

A.1.2 The solution: *DomoNet*

The *domoNet* project is born, so, to solve the big problem that pains the domotic world: the cooperation between domotic devices that belong to very different technologies (as *Konnex* and *UPnP*) and control them in the distance.

The idea is to have a *framework* able to control all the *domotic devices* placed anywhere and managed by a group of *web service* (for example, from the office it's possible to control the devices placed at home, reached phisically by a *web service*, and the devices placed to the house near the see, reached phisically by another *web service*). Each *web service* is connected phisically to the *domotic devices* and it's able to control them invoking specialized modules considering their standards. Each *web service* is also able to get a message that became from a middleware and convert it to another in order to implement the cooperation.

For realizing it, it was implemented a new XML based language called *domoML* ables to describe all the *domotic devices* (indipendently from their technology) and all the relative messages needed for their use. The *domoML* rappresents a middle language from and to which translate all messages that pass through the *web service*.

The *domoNet architecture* is composed by a set of servers (*web service*) and a possible client (for the remote control).

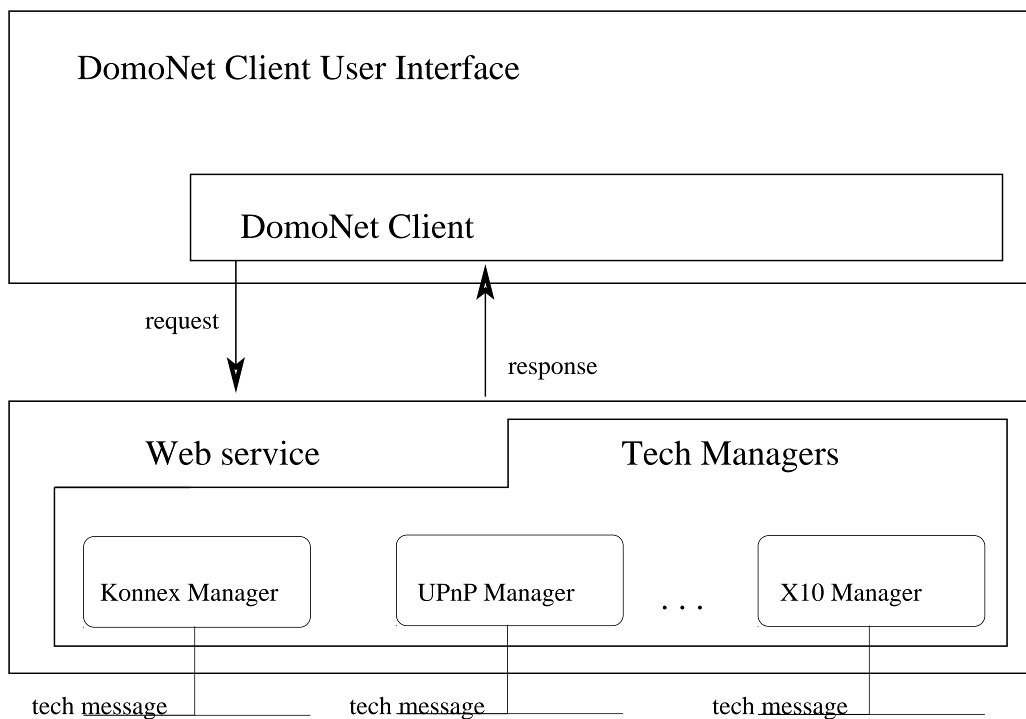

Figura A.1: The domoNet architecture

For all the application is used the *domoML* language in order to obtain in the client side and in the logic server side, the perfect abstaction of standards except for each *tech manager*

where, inside them, is used the specific protocol sintax because they must control (as drivers) the devices.

A.1.3 Terminology

- *device*: physical device belonging to a determinate technology.
- *domoML*: XML formalism used to describe and to implement functionalities of devices inside the application.
- *domoDevice*: a *domoNet* device described throught *domoML* formalism.
- *service*: a service offered by a domoDevice and that any other *domoDevice* or client can use.
- *domoMessage*: message description used in a *domoML* formalism.
- *domoAddress*: a couple composed by the *url* of the *web service* that controls the *domoDevice* and an *id* that identify the *domoDevice* inside the *web service.*
- *real address*: the real address in its technology formalism of a *device.*
- *tech manager*: module of the server side of the application that can manage a technology. Every *web service* can contains more than one domo manager of different technology.

- *tech message*: message generated that came or is made for a particular technology in order to exchange data.

A.1.4 How to install, compile and launch the application

REQUIREMENTS:

- apache-tomcat (http://tomcat.apache.org);
- axis (http://ws.apache.org/axis);
- xerces (http://xerces.apache.org/xerces2-j);
- calimero (http://calimero.sourceforge.net);
- cyberLink for Java (http//sourceforge.net/projects/cgupnpjava)

INSTALLATION:

- If not already installed, install "axis" inside the apache-tomcat copying the directory "webapps/axis" of the "axis" distribution inside the webapps directory of the apache-tomcat installation.
- Change the directory path lines (from 6 to 12) in the "build.xml" file.
- Copy the jars of the "xerces" archive into "apache-tomcat" webapps/axis/WEB-INF/lib directory.

- Copy “calimero” jars into “apache-tomcat” webapps/axis/WEB-INF/lib directory.
- Add in “apache-tomcat” webapps/axis/WEB-INF/server-config.wsdd the following lines in the services section:

```
<service name="DomoNetWS" provider="java:RPC">
 <parameter name="className"
  value="domoNetWS.DomoNetWS" />
 <parameter name="scope" value="application" />
</service>
```

If you don’t have this file, you can copy a basic configuration file at the end of this section (server-config.wsdd).

```
----------------> server-config.wsdd <---------------
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultClientConfig"
 xmlns="http://xml.apache.org/axis/wsdd/"
 xmlns:java=
 "http://xml.apache.org/axis/wsdd/providers/java"
 xmlns:handler=
 "http://xml.apache.org/axis/wsdd/providers/handler">
 <globalConfiguration>
  <parameter name="disablePrettyXML" value="true"/>
  <parameter name="dotNetSoapEncFix" value="true"/>
  <parameter name="enableNamespacePrefixOptimization"
```

```
   value="false"/>
  <requestFlow>
   <handler type=
    "java:org.apache.axis.handlers.JWSHandler">
    <parameter name="scope" value="session"/>
   </handler>
   <handler type=
    "java:org.apache.axis.handlers.JWSHandler">
    <parameter name="scope" value="request"/>
    <parameter name="extension" value=".jwr"/>
   </handler>
   <!-- uncomment this if you want the SOAP monitor -->
   <!--
    <handler type=
    "java:org.apache.axis.handlers.SOAPMonitorHandler"/>
   -->
  </requestFlow>
  <!-- uncomment this if you want the SOAP monitor -->
  <!--
  <responseFlow>
   <handler type=
    "java:org.apache.axis.handlers.SOAPMonitorHandler"/>
  </responseFlow>
  -->
 </globalConfiguration>
```

```
<handler type=
 "java:org.apache.axis.handlers.http.URLMapper"
 name="URLMapper"/>
<handler type=
 "java:org.apache.axis.transport.local.LocalResponder"
 name="LocalResponder"/>
<handler type=
 "java:org.apache.axis.handlers.SimpleAuthenticationHandler"
 name="Authenticate"/>
<service name="AdminService" provider="java:MSG">
 <namespace>http://xml.apache.org/axis/wsdd/</namespace>
 <parameter name="allowedMethods" value="AdminService"/>
 <parameter name="enableRemoteAdmin" value="false"/>
 <parameter name="className"
  value="org.apache.axis.utils.Admin"/>
</service>

<service name="Version" provider="java:RPC">
 <parameter name="allowedMethods" value="getVersion"/>
 <parameter name="className"
  value="org.apache.axis.Version"/>
</service>
<service name="DomoNetWS" provider="java:RPC">
 <parameter name="className" value="domoNetWS.DomoNetWS" />
 <parameter name="scope" value="application" />
```

```
 </service>
 <transport name="http">
  <parameter name="qs:list"
   value="org.apache.axis.transport.http.QSListHandler"/>
  <parameter name="qs:method"
   value="org.apache.axis.transport.http.QSMethodHandler"/>
  <parameter name="qs:wsdl"
   value="org.apache.axis.transport.http.QSWSDLHandler"/>
  <requestFlow>
   <handler type="URLMapper"/>
   <handler type=
    "java:org.apache.axis.handlers.http.HTTPAuthHandler"/>
  </requestFlow>
 </transport>
 <transport name="local">
  <responseFlow>
   <handler type="LocalResponder"/>
  </responseFlow>
 </transport>
</deployment>

-------------> end server-config.wsdd <---------------
```

To compile the domoNet project, have javadoc and docs, type inside the domoNet directory, the command ant.

To run a sample under linux, type: ant demo or launch launch.sh script., modifying paths (lines from 2 to 8).

A.2 DomoML

A.2.1 What *domoML* is and what it does

domoML is a language implemented throught an *XML* library based on the *Xerces library* to represent devices and interactions in a compact way abstracting them from their technology.

The *domoML*, used with this architecture, is "a middle language" from and to which translate devices and packets representation. All the logic part uses *domoML* and only the phisical iterations with device modules (in the *tech manager*) the *domoML* is translated to the *tech language* for that technology.

The *domoML* library is composed by two mainlines:

- the *domoDevice*: the part of the *domoML* that describes and creates the abstraction of the *devices*;
- the *domoMessage*: the part of the *domoML* that describes iterations from and to *devices*.

domoDevice The *domoDevice* language has the purpose to create a compact, effective, simple and complete way to represent *devices* abstracting them from the technology used.

In the *domoDevice* can be represented throught a wide and poorly deeped tree (the grammar provided high attributated tags and not many children). The tree can have maximum four levels:

1. device: opens the *domoDevice* description and gives general informations about the device as follow:

    - description: a natural language description for the *device*;
    - *id*: is a value used for take and generate the *domoDeviceId*. It identifies the *domoDevice* inside the *web service*;
    - manufacturer: the manufacturer of the *device*;
    - positionDescription: a natural language description of the location of the *device*.
    - serialNumber: the serial number (it can be composed by letters too) of the *device*.
    - tech: the technology of the original *device* represented;
    - type: the typology of the *device*;
    - *url*: is a value used for take and generate the *domoDeviceId*. It identifies the *web service* that has the *device*.

    All this fields are optional except for the *id url* and *tech* because permit to identify and use the *device*.

2. *service*: describes a service offered of the *domoDevice*. This information is used to create a *domoMessage* and to converting it to a *tech message*. For each *domoDevice* there can be more than one *service* tags. Tags for this are:

   - description: a natural language description for the *service*;
   - name: an identifier that can be useful when a *domoMessage* the will be generated and translated to *tech message*;
   - output: the *domoML.domoDevice.DomoDevice.DataType* if is exptected a return value when the corresponding *domoMessage* will be executed. This attribute must not be placed if no return value is provided.
   - outputDescription: a natural language description for the *output* attribute. If no *output* attribute is provided this attribute must not be placed;
   - prettyName:

to do: continue.

A.3 Server side

A.3.1 Setting up the *webService*

The server side of the application tries to resolve the cooperation from devices that belong to different technologies and executes

commands becaming from another server or from the client.

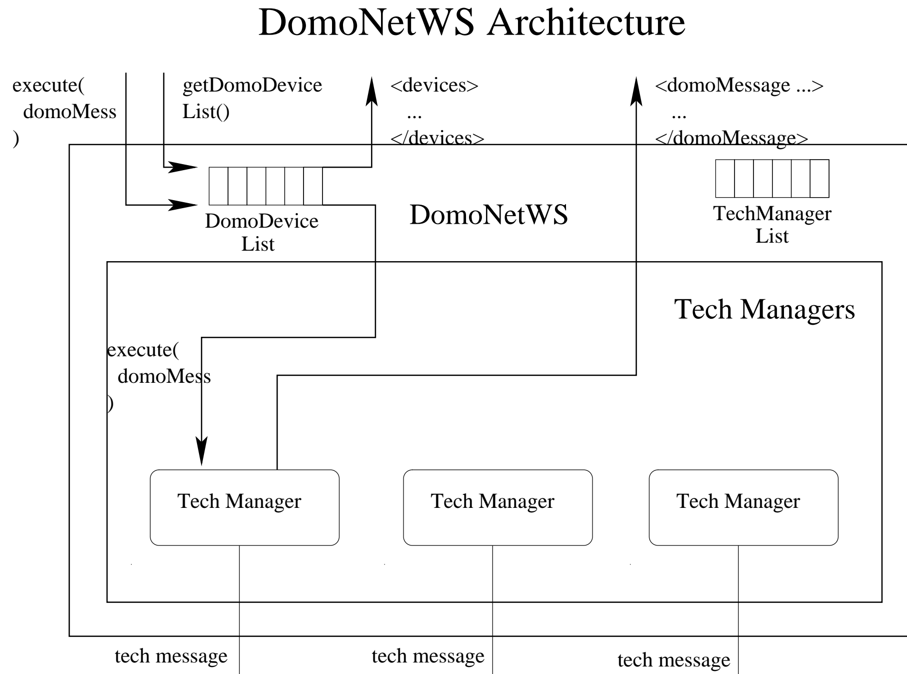

Figura A.2: The domoNetWS (server side) architecture: a general view.

Every *web service*, at startup time, setup the *tech managers* stored in the *techManagerList HashMap* and get from them the list of device currently installed. The *techManagerList* take a reference of the class implementation of a particular technology (viewed as driver or module). There can be many *tech manager* of different technologies but can't be more than one *tech managers* of the same technology in the same *web service*.

Every device recognized by every *tech manager* is converted using the *domoDevice* formalism and stored inside the *DomoDeviceList HashMap* where is indicized assigning it an unique *domoAddress* composed by the *url* of the *web service* that stores

the *domoDevice* (at startup time is empty because is not relevant here), and an *id* (a progressive positive number that permit to identify the *domoDevice* inside the *web service*).

Executing a *domoMessage*

The request to execute a *domoMessage* can came from a client application, from another *web service* or from the same *web service* when it's requested the cooperation between devices with different technologies (so, througth different *tech managers*).

When a *domoMessage* arrives in order to be executed, the first thing to do is to understeand the technology of the device that it's interested by it. To do it, using the *domoMessage* attribute, the *domoDeviceId* of the receiver of the message it's calculated in order to get its *domoDevice* description and know the technology envolved. Now, it's possible to forward the execute message to the correct *tech manager* querying the *techManagerList*. It will provide to translate it from *domoMessage* to *tech message* and to execute it. Once the message will be executed, the *tech manager* will send a *domoMessage* of failure or sucess (with value if requested) execution to the sender of the original *domoMessage*.

The cooperation between different *techManagers*

The cooperation between different *domoDevices* of different technology is the core of this work.

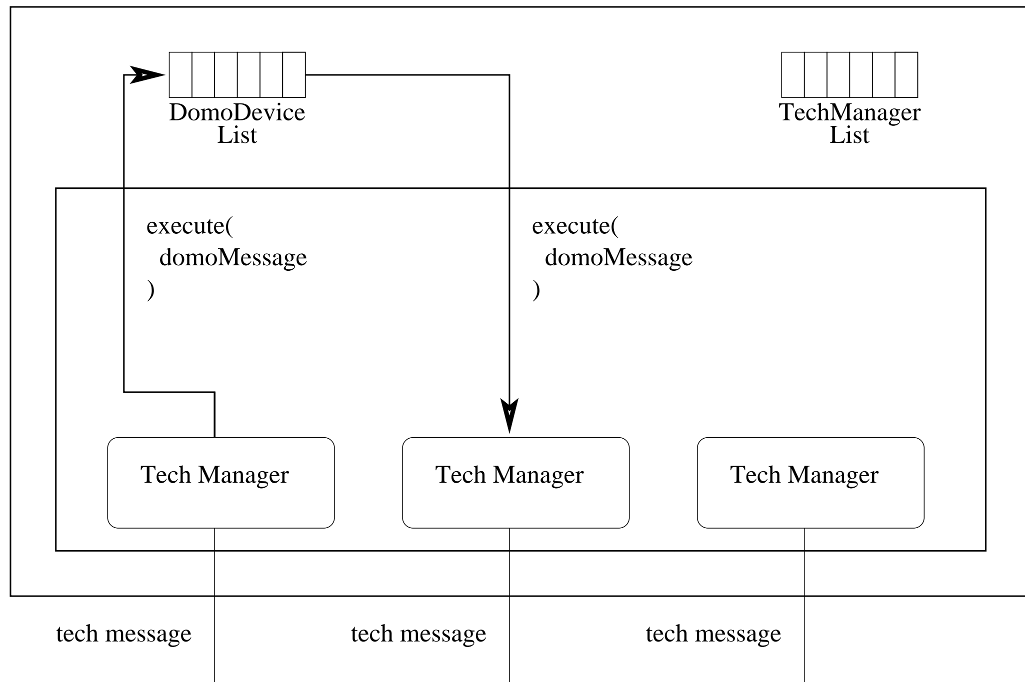

Figura A.3: The domoNetWS (server side) architecture: the cooperation

This figure shows how the cooperation between different devices works: the *tech manager* receive a *tech message* from its net and it verifies if the recipient device is a member of its technology. If not, it translates the *tech message* as *domoMessage* and roll over it to the recipient *tech manager* that provide to translate and forward it as *tech message* throught its net; else do nothing because the recipient can receive its message directly.

to do: continue.

A.3.2 The *techManagers*

A *tech manager* is a module (or driver) of the *web service* that has the function to phisically interface it with the devices of a particular technology such as *Konnex*, *UPnP*, *X10* and so on. A *tech manager* implements calls, functions and all required to implement a specific technology of devices. Its functions concerns to generate a *domoDevices* list from the available devices, translate a *domoAddress* to a*real address* and vice versa, translate a *domoMessage* to a *tech message* and vice versa.

At startup time, it must give the list of all the device currently available for that technology and initialize the eventual data structures.

Executing a *domoMessage*

When the *tech manager* receives a *domoMessage* to be executed, it must get the *real address* of the device of the receiver of the message and convert the data included in the *domoMessage* to a *tech message*.

This means that the *tech manager* must verifies if the *domoMessage* is of type *COMMAND*, understeand the action to do interpreting the attribute *message*. It must converts *input* tags (if any) considering the *datatype* and *value* attributes. At this time, the message, (now *tech message*) can be sent through the transport way for that technology.

The execution of the *tech message* can (not must) envolve

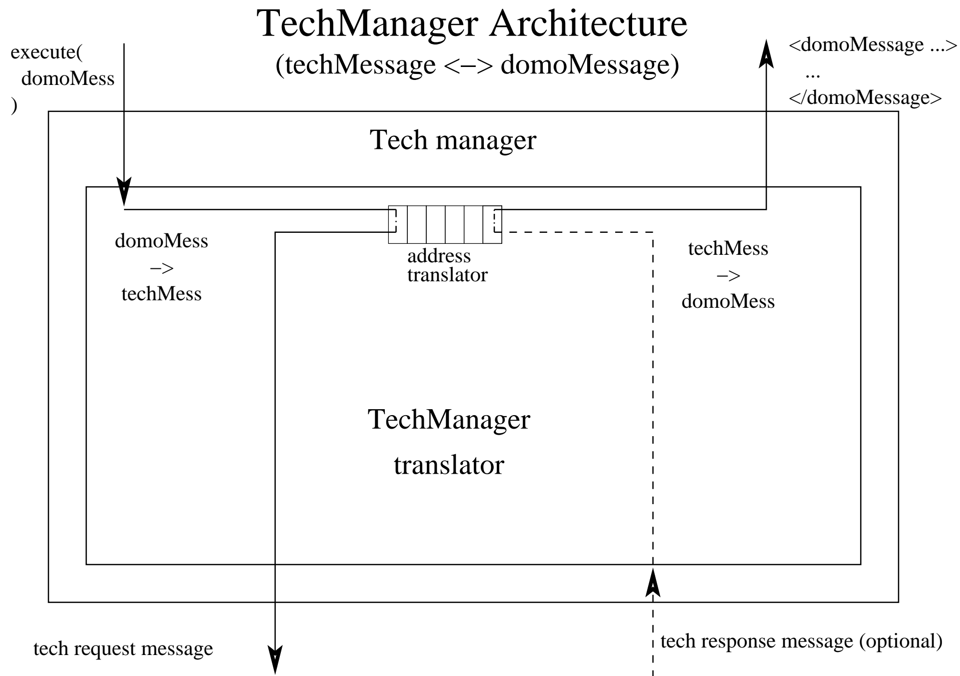

Figura A.4: The techManager architecture: the exection of a domoMessage.

others messages as example a response to the orginal message with a value or ack. If it's provided a response, it must be converted from a *tech message* to a *domoMessage* considering again the datatypes and the data. This information are stored in a *domoMessage* of type *SUCCESS* or *FAILURE*. If it is not provided a response, the *tech manager* provides an empty *domoMessage*. In any case, the response *domoMessage* can be of type *SUCCESS* or *FAILURE* (it can depend from the state of the net, from the throw of exception and so on).

To do that, so, a *tech manager* must be composed by a *domoMessage* to *tech message* and vice versa translator. To translate addresses, it uses a *DoubleHash*: a special *HashMap* that permi-

ts to use the *domoDeviceId* or the *real address* as key to obtain the other. For each *domoDeviceId* it's possible to associate more than one *real address* (useful for example for the *Konnex* technology) but for each *real address* it's possible to associate only one *domoDeviceId*.

The cooperation between different *techManagers*

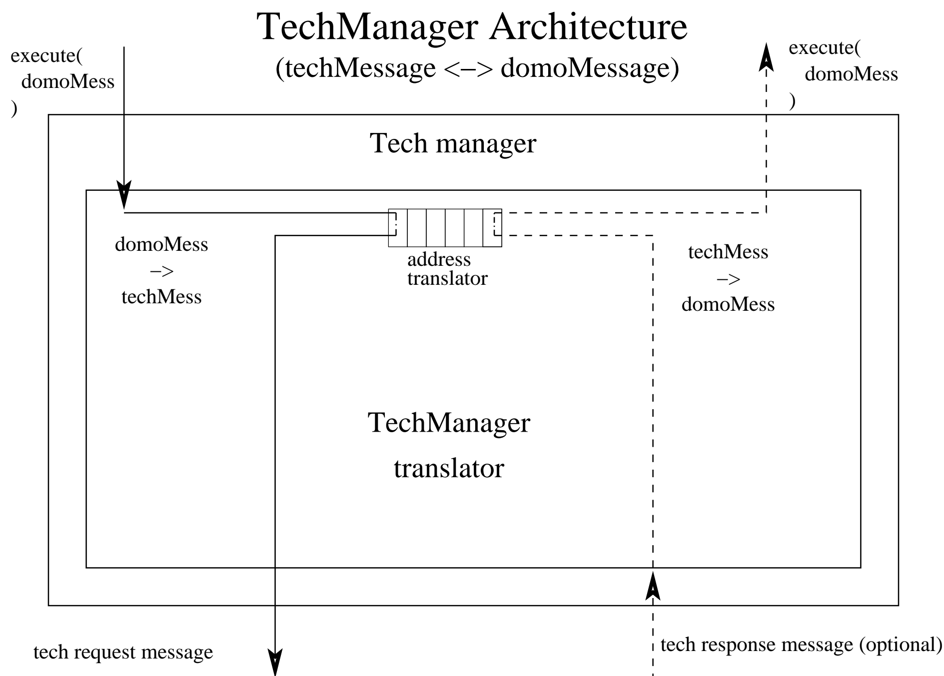

Figura A.5: The techManager architecture: the cooperation.

to do: continue.

How to implement a new *tech manager*

To implement a new *tech manager* module, it's needed to extend the *domoNetWS.techManager.TechManager* java class and

create a new instance of the new class in the *domoNetWS.DomoNetWS.managerList HashMap* using as key the identifier of the technology (it must be used one listed in *domoML.domoDevice.DomoDevice.DomoTech enum* or create new one there). The new class must have implemented the *abstract* methods of the *domoNetWS.techManager.TechManager* java class that are:

- *public void addDevice(final DomoDevice domoDevice, String address)*:
  to add a *domoML* device to the list of *domoDevice* in the *web service*; it must call the *doubleHash.add()* and *addDomoDevice()* of the owner *web service* class methods;

- *public DomoMessage execute(DomoMessage domoMessage) throws Exception*:
  execute a *domoML.domoMessage.DomoMessage*; it must be able to convert the *domoMessage* to the *tech message* and vice versa as described before;

- *public void finalize()*:
  actions to do when the manager will be closed.

Implemented *tech managers*

These are the *tech managers* just implemented and can be taken as example for new ones.

KNXManager

The *KNXManager* is the module that manages the *Konnex* technology and it is based on the *Calimero library* (http://calimero.sourceforge.net).

The constructor of the module take the *url* and the *port* number of the service that is connected to the devices.

At startup time it performs the connection and initializes the structures that manage the conversion of datatypes (exactly from *domoML.DomoDevice.domoDevice.DataType* to a *Major* and *Minor* identifiers, used to convert the *message* attribute in the *domoMessage* to an undesteandable value for the technology in the *execute* method) from string to *domoML.DomoDevice.domoDevice.DataType* and vice versa used when are loaded the devices available. After initializations, the module loads the available devices parsing an *XML* file generated by the the version number 3 of the *ETS* application[3] called *KNXConfiguration.xml*.

The emphKNXConfiguration.xml contains the description of all the functionality that the *Konnex* system can offer describing the device, the flags (I'm interested only if it's a write or read service field), the name of the service. An example of an entry of the *KNXConfiguration.xml* is:

```
<row>
```

[3]The *ETS* (EIB Tools Software) is a software distribuited by the *Konnex Association* for setting up and configure the *Konnex* system.

```
 <colValue nr="1">0/0/1 Comando Uscita A</colValue>
 <colValue nr="2">0: Comando,tasto sinistro superiore-
  Commutazione </colValue>
 <colValue nr="3">1.1.4 16196.. Pulsante 4 canali
 </colValue>
 <colValue nr="4">S</colValue>
 <colValue nr="5">-</colValue>
 <colValue nr="6">C</colValue>
 <colValue nr="7">-</colValue>
 <colValue nr="8">W</colValue>
 <colValue nr="9">T</colValue>
 <colValue nr="10">U</colValue>
 <colValue nr="11">16196
   .. Pulsante 4 canali
 </colValue>
 <colValue nr="12">16196 On-Off-Dimmer-Tapparelle-Led
 </colValue>
 <colValue nr="13">1 bit</colValue>
 <colValue nr="14">Basso</colValue>
 <colValue nr="15">0/0/1</colValue>
</row>
```

(For describing this service, it was used an *ETS* italian driver).

With this fields it's possible to know:

- from the 2nd colValue, the name of the service;

- from the 3rdth colValue (the initial part), the real address of the device;
- from the 7th colValue, if the field is readable (if set to "R" is readable);
- from the 8th colValue if the field is writeable (if set to "W" is writeable);
- from the 9th colValue, if the field can be transmetted (if set to "T" it can).
- from the 11th colValue, the name of the device that offer the service;
- from the 12th colValue, the description of the service;
- from the 15th colValue, the address group joined with the service;

A device can be described using all the *row* tags that at the 11th *colValue* have the same text. The others tags are used to describe services (one for each *row*).

to do: show the corresponding domoDevice

to do: describe how messages that come from the net are keeped.

UPnPManager

The *UPnPManager* is the module that manages the *UPnP* technology and it is based on the *Cyberlink library*

(http://sourceforge.net/projects/cgupnpjava). This library emulates the functionality of *UPnP* devices and provides already working devices. This simulation is the same as the real functionality.

At startup time it initializes the *ManagerPoint Listener* (the core of this manager) that provides to add, remove, checking hartbeat and manage the packet of devices. The *ManagerPoint Listener* also provides to initialize the data structures that manage the conversion from *domoML.DomoDevice.domoDevice.DataType* to symbols used by this technology and vice versa.

When a new device is added to this net, it announces its self giving its description and the services that can offer. This informations are stored in many *XML* files that the library just parses and gives the resulting tree and methods to cross it. At this point is very easy to get the informations regarding the device (the root of the device tree gived):

```
<device>
 <deviceType>urn:schemas-upnp-org:device:light:1
 </deviceType>
 <friendlyName>CyberGarage Light Device</friendlyName>
 <manufacturer>CyberGarage</manufacturer>
 <manufacturerURL>http://www.cybergarage.org
 </manufacturerURL>
 <modelDescription>CyberUPnP Light Device
 </modelDescription>
```

```
 <modelName>Light</modelName>
 <modelNumber>1.0</modelNumber>
 <modelURL>http://www.cybergarage.org</modelURL>
 <serialNumber>1234567890</serialNumber>
 <UDN>uuid:cybergarageLightDevice</UDN>
 <UPC>123456789012</UPC>
 <iconList>
  ...
 </iconList>
 <serviceList>
  ...
 </serviceList>
  <presentationURL>http://www.cybergarage.org
  </presentationURL>
</device>
```

and services (called *actions*):

```
<actionList>
 <action>
  <name>SetPower</name>
   <argumentList>
    <argument>
     <name>Power</name>
     <relatedStateVariable>
        Power
```

```
    </relatedStateVariable>
    <direction>in</direction>
   </argument>
   <argument>
    <name>Result</name>
    <relatedStateVariable>
       Result
    </relatedStateVariable>
    <direction>out</direction>
   </argument>
  </argumentList>
 </action>
 <action>
  <name>GetPower</name>
   <argumentList>
    <argument>
     <name>Power</name>
     <relatedStateVariable>
        Power
     </relatedStateVariable>
     <direction>out</direction>
    </argument>
   </argumentList>
  </action>
 </actionList>
```

```
  <serviceStateTable>
   <stateVariable sendEvents="yes">
    <name>Power</name>
    <dataType>boolean</dataType>
    <allowedValueList>
     <allowedValue>0</allowedValue>
     <allowedValue>1</allowedValue>
    </allowedValueList>
    <allowedValueRange>
     <maximum>123</maximum>
     <minimum>19</minimum>
     <step>1</step>
    </allowedValueRange>
   </stateVariable>
   <stateVariable sendEvents="no">
    <name>Result</name>
    <dataType>boolean</dataType>
   </stateVariable>
</serviceStateTable>
```

For each *action* is provided a set of *argument*. Each *argument* has a dataType (to be converted to *domoML.DomoDevice.domoDevice.DataType*) and if it's for input or output. Using this informations is simple to have the exactly sintax and semantic for each service.

to do: show the corresponding domoDevice

A.4 Client side

The *DomoNetClient* perfoms the control of devices in the distance regardless its technology.

It can connect to the *web service* and get the list of all the *domoDevices* (*getDeviceList()*) and sends *domoMessage* commands (*execute(dm)*) to a certain device and gets the response. All the *domoDevices* are stored in the *domoDeviceList*. It can manage more than one *web service* at time.

This is only a logical enviroment and it must be used with a graphical or textual user interface able to manage that calls and to display relative data.

DomoNetClientUI

The *DomoNetClientUI* is an *graphical user interface* for the *DomoNetClient*.

This interface permits to get a tree view of all devices avaiable. It permits to get a graphical view of a single device parsing at run time its *domoML* description. In the detailed description, it's also possible to call all specific *services* avaiable for that device.

In order to do that, it was realized a runtime interface builder that take the *domoML* device description and, parsing it, generate the needed forms.

DomoNetClientUI Architecture

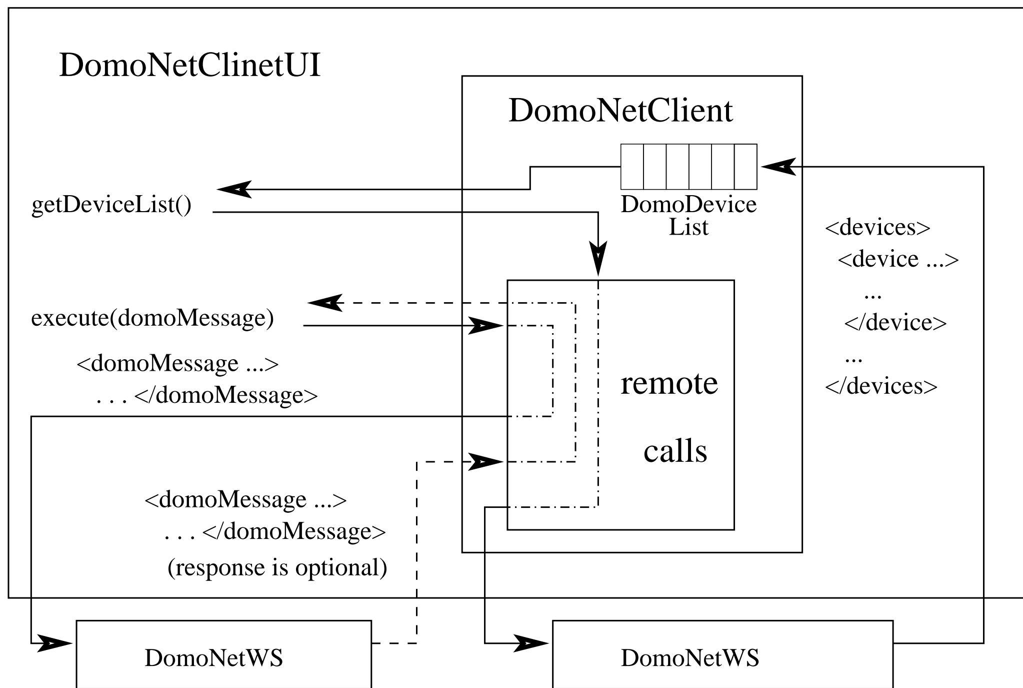

Figura A.6: The domoNetClientUI (client side) architecture

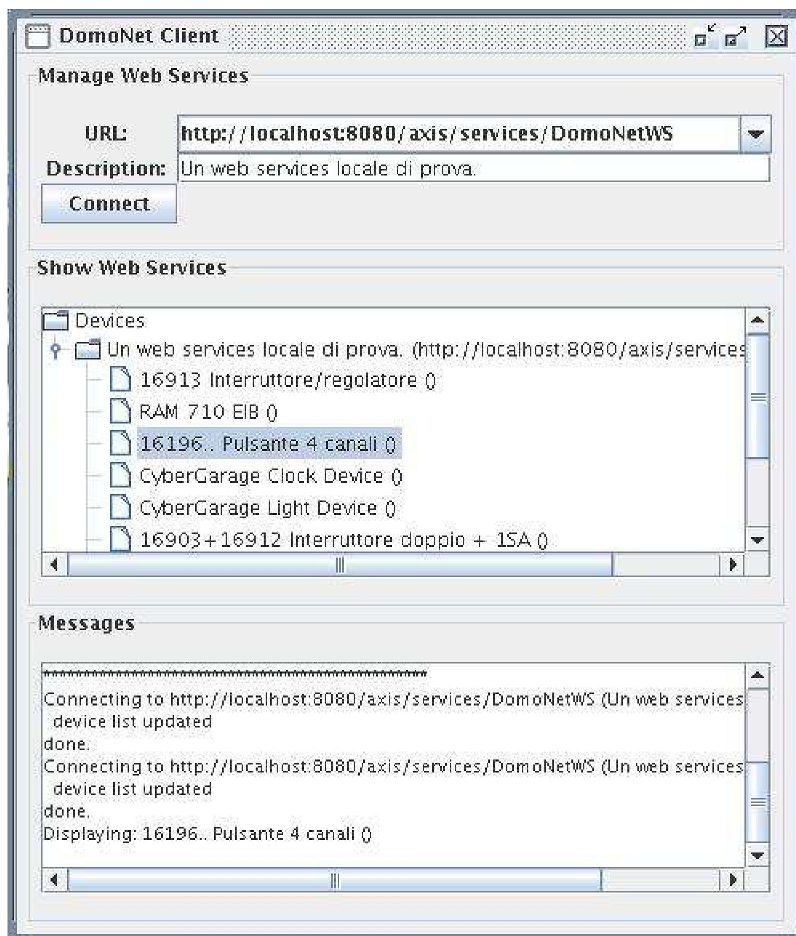

Figura A.7: The domoNetClientUI (client side) architecture

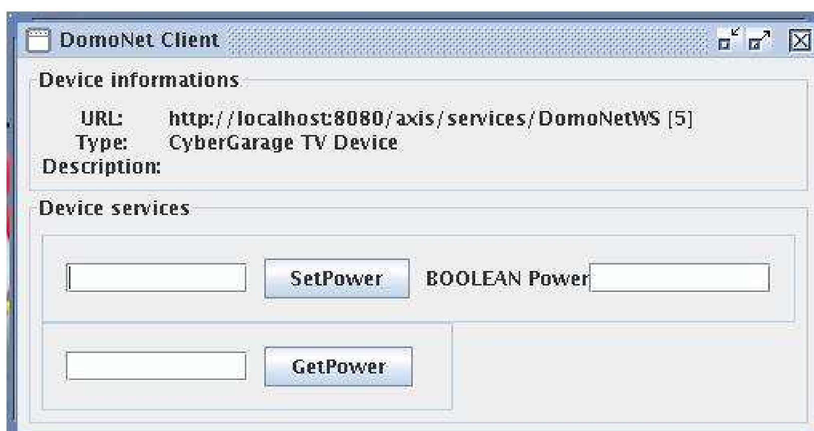

Figura A.8: The domoNetClientUI (client side) architecture

Bibliografia

- Per l'introduzione alla domotica
  - Bianchi Bandinelli R., Lucidi del corso di Domotica (CLS Informatica, a.a. 2005/06)
    www.di.unipi.it/ bandinel/domotica.html
  - Jini Community, http://www.jini.org
  - HAVi Organization, http://www.havi.org
  - Echelon Co., "Introduction to the LonWorks Platform: An Overview of Principles and Practices, v2.0", 2001
  - X10 Home Page, http://www.x10.com
- Per Konnex
  - Konnex Association, http://www.konnex.org
  - Konnex Standard System Specification: Architecture Vol.3 Part I (Konnex Association, 2001)
    (non di dominio pubblico)
- Per UPnP
  - UPnP Forum, http://www.upnp.org

  - UPnP Device Architecture 1.0 (UPnP Forum, 2003) http://www.upnp.org/resources/documents/CleanUPnPDA101-20031202s.pdf

- Per i Web Services
  - Kreger H., IBM Web Services Conceptual Architecture 1.0 (IBM Software Group, 2001)
  - Newcomer E., Understanding Web Services: XML, WSDL, SOAP and UDDI (Addison Wesley Professional, 2002)
  - Snell J., Programming Web Services with SOAP (O'Reilly, 2001)
  - Cerami E., Web Services Essential (O'Reilly, 2002)
  - http://www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf

- Per lo sviluppo
  - Tokunaga E. et al., A Framework for Connecting Home Computing Middleware (Department of Information and Computer Science Waseda University - Tokyo - IEEE, 2002)
  - Chemishkian S., Lund J., Experimental Bridge LonWorks/UPnP 1.0 (Consumer Communications and Networking Conference, 2004)
  - Papazoglou, M.P. and Georgakopolous, D. "Service Oriented Computing". In *Communications of the class ju-*

st created and obtain a relative ACM 46, 10. (2003), 42-47.

- Lee, C.E. and Moon, K.D. "Design of a Universal Middleware Bridge for Device Interoperability in Heterogeneous Home Network Middleware". CCNC Conference. (2005).
- F. Jammes, H. Smit, "Service-Oriented Paradigms in Industrial Automation", IEEE Transactions on Industrial Informatics, Vol.1 (1), pp. 62-70, February 2005.

- Per la realizzazione del prototipo
  - Mahmoud Q.H., Registration and Discovery of Web Services Using JAXR (2002)
  - Topley K., Java Web Services in a nutshell (O'Reilly, 2003)
  - Java Web Services Tutorial (Sun Microsystems, 2004)
  - McLaughlin B., Java & XML 2nd Edition: Solutions to Real-World Problems (O'Reilly, 2001)
  - Harold E.R., XML Bible 2nd Edition (John Wiley & Sons, 2001)
  - Harold E.R, Java Network Programming (O'Reilly, 2000)
  - Hyde P., Java Thread Programming: The Authoritative Solution (SAMS, 1999)
  - JNI tutorial (sun url)

- Horstmann C.S., Cornell G., Core Java 2 Volume II: Advanced Features - capitolo 11 (Prentice Hall, 1999)
- http://www.netbeans.org/kb/articles/form_getstart40.html
- http://www.devleap.it
- http://java.sun.com/developer/technicalArticles/WebServices/jaxrws/
- http://java.sun.com/webservices/tutorial.html
- http://java.sun.com/docs/books/tutorial/native1.1
- http://www.eiba.com/en/software/falcon/samples.html
- Nichols, B.A. Myers, M. Higgins, J. Hughes, T.K. Harris, R. Rosenfeld and M. Pignol, "Generating Remote Control Interfaces for Complex Appliances", Proceedings of the 15th annual ACM symposium on user interface software and technology, ACM Press, pp. 161-170, October 2002.
- EIBA Software, http://www.eiba-software.com/
- F. Furfari, C. Soria, V. Pirrelli, O. Signore, R. Bandinelli, "NICHE: Natural Interaction in Computerized Home Environment", Ercim News no. 58, pp. 66-67, July 2004.
- L. Tarrini, V. Miori, "LIGHT: XML-Innovative Generation for Home Networking Technologies", Ercim News no. 62, pp. 48-49, July 2005.

Elenco delle figure

2.1 Esempio di sistema *bus*. 11

2.2 Il modello ISO / OSI. 19

2.3 Modello architetturale del sistema *Konnex*. 24

2.4 Il sistema konnex. 25

2.5 Topologia e schema di indirizzamento della rete *Konnex*. 34

2.6 Struttura del frame *Konnex* allo strato Data-Link. 50

3.1 *web service*: entità, operazioni, artefatti. 57

3.2 Stack protocollare dei *web service*. 62

3.3 Il formato del messaggio SOAP. 69

3.4 Struttura del documento *WSDL*. 74

4.1 Rappresentazione dei middleware domotici. 79

4.2 Il framework domoNet. 83

4.3 L'architettura domoNet 84

6.1 L'architettura del domoNetWS (web service). . . 97

6.2 L'architettura del domoNetWS (web service): la cooperazione. 99

6.3 L'architettura del domoNetWS (web service): la cooperazione tra diversi *web service*. 101
6.4 L'architettura del techManager: l'esecuzione di un domoMessage. 102

7.1 L'architettura del domoNetClient. 105

8.1 Pannello *Konnex*. 108
8.2 I dispositivi domotici *UPnP* simulati utilizzati. . . 110
8.3 Snapshot del domoNetClientUI (finestra principale). 137
8.4 Snapshot del domoNetClientUI (finestra dei servizi 1). 138
8.5 Snapshot del domoNetClientUI (finestra dei servizi 2). 139
8.6 Pulsantiera *Konnex* a 4 canali. 143
8.7 Accensione lampada *UPnP* alla pressione del tasto della pulsantiera *Konnex*. 144
8.8 Dispositivo di controllo remoto *UPnP*. 145
8.9 Accensione lampade *Konnex* e *UPnP*. 146

A.1 The domoNet architecture 157
A.2 The domoNetWS (server side) architecture: a general view. 167
A.3 The domoNetWS (server side) architecture: the cooperation 169

A.4 The techManager architecture: the exection of a domoMessage. 171
A.5 The techManager architecture: the cooperation. . 172
A.6 The domoNetClientUI (client side) architecture . 182
A.7 The domoNetClientUI (client side) architecture . 183
A.8 The domoNetClientUI (client side) architecture . 183

Elenco delle tabelle

2.1 Regolamentazione sull'assegnazione delle frequenze. 14
2.2 Velocità di trasmissione. 16
2.3 Caterorie applicative. 16