UNIVERSITÀ DI PADOVA  FACOLTÀ DI INGEGNERIA

TESI DI LAUREA

Realizzazione di un sistema di gestione remoto per forni di cottura professionale

Relatore: Prof. Stefano Vitturi

Correlatore: Prof. Ivan Cibrario Bertolotti

Laureando: Alberto Sbeghen

Corso di Laurea SPECIALISTICA in INGEGNERIA INFORMATICA

A.A. 2010/2011

Questo testo è stato scritto con il software freeware LaTeX

L'autore può essere contattato all'indirizzo di posta elettronica:
alberto.sbeghen@gmail.com

Stampato in proprio il 18 Aprile 2011

A mio padre

Sommario

Il testo descrive il lavoro svolto durante l'attività formativa di tirocinio presso UNOX S.p.A. per la realizzazione di un sistema di gestione remoto per i forni professionali di cottura prodotti dall'azienda.

Lo studio del sistema, chiamato UNOX Remote Control (URC), inizia con un'attenta analisi del contesto operativo e di tutte le problematiche connesse all'interfacciamento del nuovo sistema ai forni UNOX; sulla base di questa trattazione è stato stabilito l'insieme delle specifiche tecniche.

Il testo continua con una descrizione sommaria dell'architettura del sistema, introducendo le caratteristiche di base di ognuno dei tre elementi principali.

Il *servizio di sistema*, un applicativo dedicato alla gestione dei dati prelevati dai forni, permette di configurare URC presso l'utilizzatore finale, in modo da rispecchiare la classica struttura aziendale ad albero. Ogni nodo dell'albero, ovvero ogni copia del servizio di sistema, viene installato in una specifica sede del cliente. L'insieme dei nodi crea così una rete di interconnessione, che permette di avere il controllo dell'intero sistema direttamente dalla sede principale del cliente.

Il secondo elemento, l'applicazione di *interfaccia web-based*, offre al cliente un pannello di controllo dinamico, consultabile direttamente da web browser, attraverso il quale è possibile controllare l'esecuzione di tutte le attività del sistema.

Infine il testo presenta la *scheda elettronica* realizzata da UNOX (provvista di interfaccia Ethernet) per la raccolta dei dati dei forni e descrive i test di collaudo preliminari eseguiti sull'intero sistema.

Ringraziamenti

Questo piccolo spazio rappresenta l'occasione più grande che ho per poter esprimere i miei più sinceri ringraziamenti a tutte le persone che mi sono state vicine in questi anni. Lasciando i loro nomi in queste poche righe voglio così coronare il mio successo, che ho potuto raggiungere anche grazie a loro.

Ringrazio i miei genitori, Daria e Rino, che in questi anni mi hanno sempre supportato, anche nei momenti in cui alcune mie scelte avrebbero inevitabilmente influito sulla durata del mio percorso di studi. Hanno saputo guidarmi rimanendo spettatori attenti, proprio come avrei voluto.

Un grazie a Sara, Francesco e nonna Evelina per la loro premura e per il sostegno che mi hanno dimostrato in tutti questi anni. Un grazie al piccolo Lorenzo per i mille sorrisi e per i mille "Zio, mamàn!" che anche in questi ultimi giorni mi hanno distratto da tutte le pressioni.

Un grazie sincero a tutti gli Amici, con la A maiuscola, perché sono pienamente convinto che risultati come questo siano frutto, oltre che di tanto impegno, di momenti di spensierata e sana allegria. Grazie davvero.

Ringrazio il Prof. Stefano Vitturi, perché i suoi consigli e la sua piena disponibilità si sono rivelati davvero preziosi.

E il mio ultimo grazie va a Giulia, a lei che è stata ed è ogni giorno la mia più bella e inaspettata sorpresa. Mi sento davvero fortunato ad averla qui al mio fianco e sono davvero felice di condividere con lei questo splendido traguardo.

Pove del Grappa, il 16/04/2011

A.S.

Indice

Elenco delle figure

Capitolo 1

UNOX Remote Control

La diffusione di dispositivi e prodotti che integrano una logica elettronica di controllo è in continuo aumento, e le esigenze imposte dalle moderne dinamiche aziendali fanno sì che i sistemi di supervisione e controllo remoti siano un argomento di estrema attualità. Se fino a qualche tempo fa le attività di controllo e manutenzione dei dispositivi potevano essere svolte manualmente, oggigiorno, un metodo di lavoro in ottica di qualità totale impone che tali attività debbano essere compiute in pochissimo tempo e senza bisogno di raggiungere fisicamente il dispositivo.

I vantaggi offerti da questa tipologia di sistemi sono molteplici e possono essere riassunti principalmente nei seguenti punti:

- individuazione pressoché immediata dei guasti e dei malfunzionamenti;
- abbattimento dei costi di manutenzione;
- possibilità di reportistica e analisi dei dati in tempo reale;
- capacità di reazione immediata agli eventi;
- maggiore soddisfazione degli utilizzatori e/o della clientela.

Se pensiamo, ad esempio, ad un impianto per il controllo remoto dei dispositivi presenti in una galleria, possiamo tradurre quanto appena detto nel modo seguente:

- il sistema può segnalare ad un addetto il malfunzionamento di segnalatori luminosi, sensori antincendio, aspiratori, ecc;
- non è necessario che gli addetti alla manutenzione effettuino sopralluoghi continui per verificare lo stato dell'impianto, poiché possono intervenire solo nel caso in cui il sistema rilevi delle anomalie;
- i livelli di inquinamento, temperatura, umidità e movimenti del terreno possono essere rilevati in tempo reale e archiviati nel tempo;
- il sistema può lanciare un allarme in caso di guasto, o in caso di incidenti, permettendo di identificare con precisione il tratto di strada interessato dall'evento;

- gli automobilisti godranno di una maggiore sicurezza e di un migliore stato del tratto stradale.

Pertanto, per un'azienda che opera nell'ambito della produzione industriale e che progetta dispositivi dotati di una qualche forma di controllo, integrare nella propria gamma di prodotti un sistema di gestione remota si traduce spesso in una scelta obbligata.

UNOX, azienda del Padovano che produce forni di cottura professionale, ha deciso di dotare i propri prodotti di un innovativo sistema di gestione remoto: UNOX Remote Control.

1.1 Profilo aziendale di UNOX

Il presente elaborato documenta l'esperienza di tirocinio, che si è tenuta in collaborazione con UNOX S.p.a., volta alla progettazione e allo sviluppo di un sistema di gestione remota dei forni prodotti.

Figura 1.1: Il logo di UNOX.

L'azienda, nata nel 1990 e con sede a Vigodarzere (PD), grazie ad un continuo processo di ricerca e di innovazione, si è rapidamente imposta nel mercato internazionale come azienda leader nella produzione di forni professionali nel settore panificazione, pasticceria e gastronomia.

UNOX è un'azienda relativamente giovane in quanto nasce nel 1990 a Vigodarzere (PD) come azienda artigianale, produttrice di semplici forni a convezione che si contraddistinguevano per il prezzo basso ed una buona qualità di fabbricazione. Per i primi anni di vita dell'azienda il mercato di riferimento è stato quello spagnolo, che tipicamente si caratterizza per essere particolarmente attento al prezzo di vendita. Questo mercato ha contribuito enormemente alla crescita dell'azienda che ha dovuto prestare moltissima attenzione al contenimento dei costi di produzione, esperienza che si sarebbe rilevato essere di fondamentale importanza nel futuro dell'impresa.

Nei primi anni di vita UNOX non prestò particolare attenzione al mercato italiano che si caratterizzava per essere molto frazionato tra innumerevoli piccoli fornitori locali.

In seguito (1994) UNOX iniziò ad esportare i propri prodotti anche in altri stati del centro Europa; in particolare l'ingresso nel mercato tedesco si rivelò di fondamentale importanza, in quanto le caratteristiche di questo mercato erano completamente diverse da quelle conosciute fino a quel momento. In quegli anni i principali produttori di forni professionali al mondo erano tedeschi e l'attenzione riservata alla qualità progettuale, realizzativa e di cottura era assolutamente diversa da quella dimostrata dal mercato spagnolo.

La sfida che UNOX colse fu quella di perseguire un notevole incremento della qualità rimanendo il più competitiva possibile in termini di prezzo. A tal fine UNOX ha implementato la tecnologia produttiva di tipo "lean production" su derivazione Toyota, ha cercato di ridurre al minimo la difettosità ed ha sviluppato i nuovi prodotti in modo sempre più modulare per garantire un'altissima standardizzazione della componentistica.

Il successo delle scelte compiute si è concretizzato con l'ingresso nel mercato giapponese, conosciuto per un'attenzione al dettaglio ed alla qualità produttiva ancora superiore. In quegli anni l'azienda dovette far fronte ad un'ulteriore sviluppo e perfezionamento dei propri prodotti.

Oggi UNOX è il primo produttore mondiale in termini di forni venduti ogni anno (circa 60.000 pezzi) ed è presente con i suoi prodotti in 85 paesi al mondo.

Più di cinquanta brevetti registrati e una tensione costante al miglioramento ne fanno un'azienda che può portare sempre oltre gli standard qualitativi di settore, raccogliendo ogni giorno sfide e idee lanciate dal mercato.

Figura 1.2: La nuova sede di UNOX, che sarà pronta a Giugno 2011.

Proprio grazie ad un'ottica di lavoro che mira al continuo miglioramento, UNOX ha di recente raccolto i pareri della clientela, la quale, soprattutto nel caso di grandi catene di ristorazione, necessita di un sistema che permetta la supervisione e il controllo remoto dello stato dei forni installati nei diversi punti vendita.

Da qui l'idea di realizzare un dispositivo, acquistabile come optional, che permetta al cliente di colmare anche questa nuova esigenza.

1.2 La gamma prodotti UNOX e l'introduzione del nuovo sistema di gestione remoto

I forni professionali UNOX sono pronti a soddisfare ogni esigenza in modo completo e versatile. Ogni prodotto è stato ideato e sviluppato pensando al cliente che lo utilizzerà: le caratteristiche che contraddistinguono i prodotti UNOX sono qualità, competitività e semplicità di utilizzo.

La gamma prodotti è costituita da 5 diverse famiglie, ognuna delle quali dedicata ad una particolare clientela:

- ChefTop: forni di cottura dedicati a risotoranti, hotel e supermercati;
- BakerTop: forni dedicati a panifici e pasticcerie;
- LineMiss: forni dedicati a bar, dalle dimensioni contenute;
- LineMicro: forni adatti a piccole produzioni e per la cottura di prodotti surgelati;
- SpidoCook: piastre di cottura per il settore ristorazione e bar.

Figura 1.3: La famiglia di forni ChefTop nelle diverse taglie di produzione.

Le famiglie di prodotti per le quali UNOX intende sviluppare il sistema di controllo remoto sono le prime due, ovvero tutti i forni di fascia alta dedicati al settore della grande distribuzione. E' proprio questa categoria di clienti a sentire questa specifica esigenza, in quanto nella maggior parte dei casi la diffusione capillare sul territorio dei vari centri di distribuzione non permette di avere riscontri effettivi ed immediati sullo stato dei forni e sull'andamento della produzione delle pietanze. In questi casi, infatti, ottenere report che riassumano il numero di cotture avviate, o dati sulla qualità di cottura degli apparecchi, diventa improponibile.

Un sistema elettronico in grado di dialogare direttamente con le unità di controllo interne al forno permette invece di esportare (e, come vedremo in seguito, anche di importare) in modo veloce, preciso ed affidabile, tutti i dati di interesse, i quali possono essere archiviati ed aggregati in modo opportuno al fine di ottenere un'immagine della reale situazione delle vendite.

Inoltre, un sistema di questo tipo può consentire una più rapida individuazione dei guasti alle diverse parti dei forni, consentendo all'utilizzatore di mettere in atto delle politiche di manutenzione del proprio parco macchine che mirano a:

- rispondere prontamente in caso di guasto;
- avere un risparmio sui programmi di manutenzione programmata;
- avere un miglioramento della qualità dei propri prodotti cotti con forni UNOX.

Nell'ultimo caso, infatti, si pensi all'ipotesi in cui un forno installato in un determinato centro di distribuzione funzioni ad una potenza inferiore alla norma; chiaramente se l'inconveniente provoca dei problemi evidenti in fase di cottura l'anomalia può essere riscontrata anche dal personale di cucina. Se il problema, invece, porta ad un abbassamento della qualità di cottura del prodotto senza però comprometterne l'aspetto e le caratteristiche esterne l'anomalia verrebbe riscontrata solamente dagli acquirenti finali senza possibilità alcuna da parte della catena di distribuzione di individuare il problema per tempo.

Altri scenari che aiutano a capire l'importanza che può assumere tale sistema saranno più chiari in seguito. In ogni caso ciò che risulta evidente è che UNOX Remote Control può far fronte ad una grande varietà di esigenze diverse.

1.3 Obiettivi dell'attività di tirocinio

L'esperienza di tirocinio in UNOX si è avviata, come è già stato evidenziato, per espressa volonta dell'azienda di progettare e sviluppare il nuovo sistema di gestione remota dei forni.

Va sottolineato che l'attività di tirocinio è iniziata durante le fasi iniziali di avvio del progetto, pertanto tutto il lavoro svolto in questi mesi rientra nelle attività che sono state oggetto dall'esperienza di tirocinio.

Il lavoro in azienda è stato sviluppato all'interno del reparto tecnico-elettronico, sotto la guida e la supervisione dei responsabili interni ad UNOX.

Gli obiettivi dell'esperienza, che sono stati delineati e concordati fin da subito da tutte le parti coinvolte nel progetto, possono essere riassunti nei seguenti punti:

- analisi e studio del contesto, delle problematiche e delle esigenze della clientela;
- ricerca delle opportunità in campo ingegneristico per la risoluzione del problema;
- studio dell'architettura hardware e software del sistema e scelta dei linguaggi, degli ambienti di sviluppo e degli applicativi più adatti allo scopo;
- implementazione delle funzioni di base del sistema al fine di predisporre la realizzazione di tutte le funzioni necessarie;
- realizzazione e test di un prototipo funzionante del sistema di gestione remota per la gestione dei programmi di cottura dei forni.

L'attività ha avuto una durata di circa 5 mesi ed ha portato alla realizzazione di tutti gli obiettivi prefissati. I risultati raggiunti in questi mesi permetteranno ad UNOX di procedere con la seconda fase di sviluppo del sistema, la quale prevede l'implementazione di tutte le funzioni e le fasi finali di collaudo dell'intero sistema.

Capitolo 2

Analisi del contesto e definizione delle specifiche

L'approccio che si adotta nell'affrontare una qualsiasi tipologia di problema deve essere mirato all'individuazione di tutte le problematiche connesse alla risoluzione del caso, le quali emergono da un'attenta analisi del contesto. Nel caso specifico il contesto non contempla solo le diverse esigenze della clientela, ma poiché il dispositivo che si dovrà progettare dovrà essere interfacciato a dei prodotti già esistenti, occorre prestare estrema attenzione anche a tutti i dettagli connessi ai forni prodotti dall'azienda.

Per questo motivo il presente capitolo parte proprio da un'analisi generale della struttura dei forni UNOX, per poi passare in rassegna le caratteristiche legate all'organizzazione dei clienti, allo scopo di definire le specifiche che il sistema dovrà soddisfare.

2.1 Struttura di un forno UNOX

I forni professionali si differenziano dai forni destinati al settore consumer per diversi aspetti.

In primo luogo sono dotati di una scocca e di una componentistica molto resistente, capace di sopportare grandi carichi di lavoro; la camera di cottura e tutte le parti interessate dallo sporco sono realizzate con materiali e forme che facilitano le operazioni di pulizia.

Ma se materiali e forme non cambiano sensibilmente la qualità delle pietanze prodotte, ciò che fa veramente la differenza è racchiuso in un complesso sistema di regolazioni automatiche e misurazioni.

I forni UNOX, oltre a regolare la normale temperatura di cottura, sono provvisti di sonde atte a monitorare l'umidità interna alla camera. Questa caratteristica consente di ottenere delle cotture uniformi, anche in variazione delle caratteristiche ambientali o delle quantità di prodotto da cuocere, garantendo così la ripetibilità del risultato.

Un altro importante aspetto sta nell'utilizzo di una o più sonde di temperatura che vanno applicate "al cuore" del prodotto da cuocere, le quali consentono quindi di monitorare il reale grado di cottura del cibo.

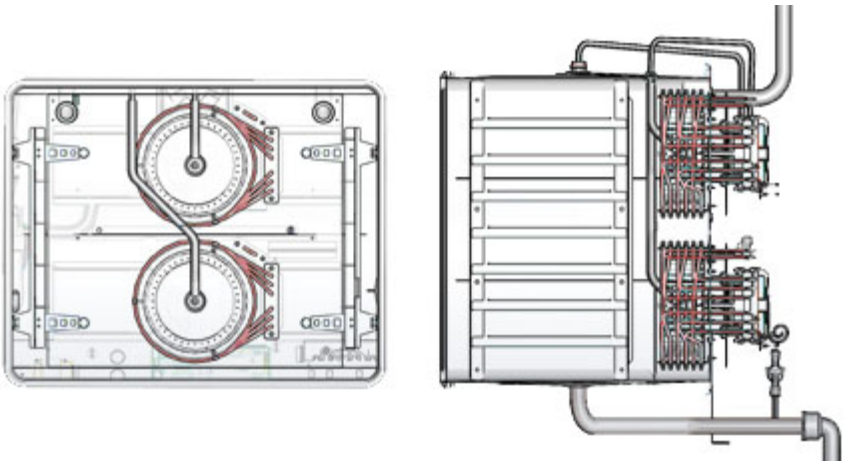

Figura 2.1: Spaccato interno di un forno UNOX.

Infine anche una corretta ventilazione del forno è una componente indispensabile per la cottura dei cibi. L'aria è il veicolo di trasmissione del calore, e deve essere correttamente indirizzata all'interno del forno per evitare che i risultati di cottura siano diversi a seconda della posizione del cibo sulla teglia (orizzontale) e dell'altezza delle teglie (verticale). Per questo in tutti i forni UNOX vengono studiati nel dettaglio i flussi d'aria interni alla camera di cottura.

2.1.1 L'elettronica di bordo, le colonne di forni e il bus Modbus

Tutte le sonde e gli organi elettromeccanici a cui si è fatto riferimento finora hanno bisogno di un sistema di controllo che ciclicamente effettui le letture degli input (sonde) per regolare il funzionamento dei dispositivi di uscita (resistenze, ventole, ecc).

Nei forni UNOX tutto questo viene svolto da una o più schede elettroniche di controllo provviste di microcontrollore e di tutta la componentistica necessaria all'interfacciamento con le diverse periferiche, sia di input che di output.

Nella maggior parte dei casi i microcontrollori usati hanno architettura ARM e consentono di mettere in atto degli elaborati programmi di cottura.

Un'altra peculiarità dei forni UNOX sta nelle diverse possibilità di installazione e configurazione. I forni, infatti, oltre che essere utilizzati come dispositivi a sé stanti, possono essere installati in colonne (solitamente compeste da due forni), con un forno posto in basso ed uno fissato più in alto.

Oltre ai forni, solitamente nelle installazioni più complesse si trovano anche altri accessori, come abbattitori (dispositivi per il raffreddamento/congelamento dei cibi) e mantenitori (forni a cottura lenta). Anche questi apparecchi montano al loro interno

delle schede elettroniche di controllo le quali sono in grado di dialogare con le unità forno.

Ogni forno prevede l'installazione di alcuni optional. Primo fra tutti il sistema Rotor Klean, che attraverso l'impiego di un girante a pressione permette la pulizia totale della camera di cottura. Si tratta di un dispositivo che consente la gestione di cicli di lavaggio utilizzando acqua, detergente e brillantante. I cicli di lavaggio alternano fasi ad alta temperatura, risciacqui con acqua, detergente, brillantante ed asciugatura finale.

Tutti i dispositivi presenti in una colonna forni UNOX effettuano uno scambio dati utilizzando un unico bus, seguendo le specifiche dello standard Modbus. Lo standard Modbus è un protocollo di comunicazione industriale nato nel 1979, che a livello fisico basa il proprio funzionamento su un canale di tipo RS232[1]. Sebbene lo standard abbia compiuto 30 anni viene ancora molto utilizzato in ambito industriale per i seguenti motivi:

- è versatile e royalty-free;
- può essere implementato facilmente con costi molto bassi;
- a livello fisico sfrutta uno stardard supportato da tutti i microcontrollori di fascia medio/alta presenti in commercio.

Considerando le dimensioni che vengono solitamente raggiunte da una colonna forni UNOX risulta estremamente comodo poter contollare tutti i diversi dispositivi attraverso un'unica maschera comandi. Pertanto, nel caso di una colonna con due unità di cottura, display e comandi di uno dei due forni vengono "spenti" ed il controllo dell'intero impianto di cottura viene delegato alla maschera comandi del secondo forno. In realtà la scheda inattiva risulta completamente funzionante e continua ad occuparsi del controllo dei dispositivi interni al proprio forno, ma l'avvio, l'arresto e il controllo dei diversi programmi di cottura avviene esclusivamente per mezzo della maschera comandi attiva.

Ecco che il bus, come vedremo in seguito, oltre a rappresentare un importante mezzo di comunicazione tra le diverse periferiche (di un singolo forno o di una colonna) è anche l'unico punto ci accesso che il sistema di controllo remoto utilizzerà per scambiare dati con le varie schede interne ai forni.

2.1.2 La maschera comandi e i programmi di cottura

L'interfaccia utente dei forni UNOX è costituita da una maschera comandi (chiamata ChefTouch) che presenta diversi display a 14 segmenti, indicatori a LED ed una serie di pulsanti, tutti di tipo touch. L'adozione della tecnologia sensibile al tocco consente di alloggiare tutti i dispositivi elettronici di interfaccia sotto ad uno spesso strato di vetro, il quale, oltre che facilitare le operazioni di pulizia, garantisce la massima protezione dallo sporco.

La maschera comandi, rappresentata in Figura 2.2, permette di impostare e controllare tutti i parametri di cottura. In particolare, permette all'utilizzatore di salvare

[1] esistono altre versioni del protocollo Modbus basate su interfacce Ethernet e RS485.

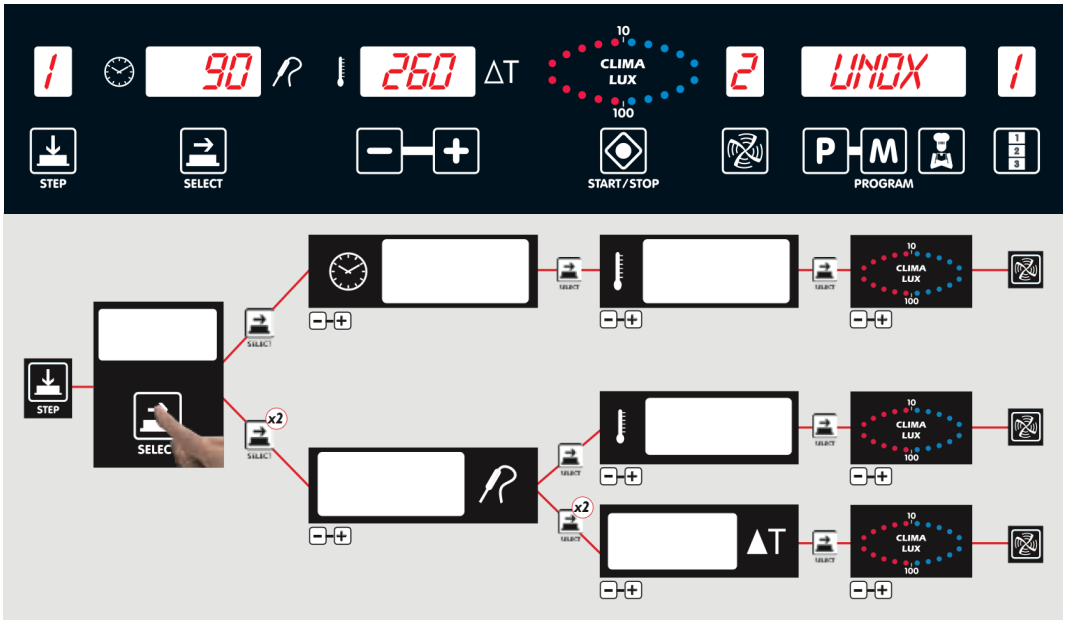

Figura 2.2: La maschera comandi ChefTouch e le relative istruzioni di utilizzo.

e avviare svariati programmi di cottura, alcuni dei quali vengono pre-impostati da UNOX, mentre altri possono essere configurati dall'utente stesso. Per ognuna delle due tipologie di programma sono presenti 100 locazioni di memorizzazione.

Ogni singolo programma di cottura è costituito da 10 diverse fasi, ognuna delle quali si conclude al raggiungimento di un particolare stato. I 10 step si dividono in un primo step di preriscaldamento (il quale può terminare dopo il raggiungimento di una specifica temperatura di camera o dopo una differenza ΔT) e 9 step di cottura normali. Ognuno dei 9 step successivi al preriscaldamento permette di configurare:

1. la condizione di fine cottura, che può verificarsi dopo che è trascorso un intervallo di tempo prefissato, oppure dopo che è stata raggiunta una specifica temperatura cuore;

2. una temperatura camera in alternativa ad un ΔT come parametro da mantenere costante durante la cottura;

3. la modalità Climalux desiderata per il secco o l'umidità (in percentuale) e la velocità delle ventole.

Ogni programma di cottura viene contraddistinto da un nome e dalla maschera comandi può essere richiamato attraverso il numero associato alla sua posizione di memorizzazione.

Solitamente ogni programma di cottura assume il nome dell'alimento o del preparato che dev'essere cotto o riscaldato. In Tabella 2.1 vengono riportate, a titolo esemplificativo, le diverse fasi che portano ad una cottura ottimale di un pollo.

Oltre ai programmi di cottura "standard", sono previsti dei programmi di cottura detti ACM, i quali permettono di riprodurre le fasi di cottura di una pietanza sulla

Fase N.	Durata	Temperatura camera	Umidità
0	preriscaldamento	150° C	0%
1	15 minuti	150° C	30%
2	15 minuti	175° C	10%
3	8 minuti	195° C	0%

Tabella 2.1: fasi di cottura del programma POLLO

base dei dati provenienti da una cottura campione. In sostanza per salvare un programma di tipo ACM occorre eseguire la cottura di un cibo, ad esempio avviando un programma di cottura standard. Durante l'intera cottura l'elettronica di bordo registra l'evolversi di tutti i parametri di funzionamento (temperatura, potenza ventole, umidità, ecc) e li salva come dati di riferimento. In seguito, l'avvio del programma ACM così salvato farà si che il forno, anche in presenza di condizioni di lavoro differenti (più o meno cibo, condizioni delle materie prime differenti, ecc) riproduca le fasi di cottura allo stesso modo, garantendo la ripetibilità del risultato.

Come vedremo in seguito, tutte queste informazioni torneranno utili per implementare le procedure di importazione/esportazione dei programmi di cottura attraverso il sistema di gestione remota.

Naturalmente i programmi di cottura all'interno del forno vengono salvati in una memoria non volatile e solo i programmi di cottura di tipo utente risultano modificabili; quelli pre-impostati, al contrario, risultano bloccati in scrittura (possono solo essere richiamati per l'avvio).

Va evidenziato che anche i dispositivi accessori, come abbattitore e mantenitore, sono dotati di programmi di cottura; i parametri di questa tipologia di programmi possono differire rispetto a quelli relativi ai forni, ma ciò che è importante è che internamente alle schede di controllo questi assumono lo stesso formato di salvataggio.

2.1.3 Numero di serie dei prodotti e identificazione delle periferiche connesse al bus

Altra caratteristica che dovrà essere considerata durante la progettazione del sistema riguarda l'identificazione dei dispositivi prodotti da UNOX.

Purtroppo le versioni dei forni attualmente in produzione non dispongono di numeri di serie univoci, o per meglio dire, questi non possono essere recuperati a livello di elettronica (nella memoria delle schede di controllo prodotte il numero di serie non viene inserito). Pertanto, a meno che non si modifichi il firmware interno ai microcontrollori, non si potrà implementare un meccanismo di identificazione dei prodotti a livello globale[2].

Inoltre, nelle versioni attuali dei forni, non esiste un modo per "conoscere" o recuperare l'elenco dei dispositivi connessi al bus. Per cui, in previsione di implementare una funzione che permetta di prelevare tali informazioni (anche da remoto)

[2]tale procedura potrebbe consentire ad UNOX di mettere in atto, ad esempio, delle politiche di controllo in materia di qualità per l'identificazione di guasti connessi a determinati lotti di produzione.

per recuperare la struttura dei vari impianti, occorrerà considerare anche questo fatto.

Per concludere va evidenziato che, come già detto in precedenza, ogni dipositivo collegato al bus viene contraddistinto da un determinato indirizzo. Ma tale indirizzo non viene fissato a priori da UNOX (ad es. tutti i mantenitori hanno indirizzo 5), poiché viene stabilito in fase di montaggio dell'impianto. Pertanto l'elettronica di bordo non potrà fare affidamento sugli indirizzi Modbus per dialogare con una particolare periferica.

2.2 Gli utilizzatori finali del sistema

L'analisi delle caratteristiche legate agli utilizzatori finali riveste una grande importanza, in quanto consente di mettere a punto un sistema che aderisca a tutte le loro esigenze. Inoltre occorre fare in modo che l'introduzione del nuovo sistema non comporti spese eccessive legate alla predisposizione dell'infrastruttura necessaria, sia a livello sistemistico sia a livello di personale. Anche il suo utilizzo dovrà risultare il più semplice possibile, celando all'utente tutte le complicazioni tecniche insite nel sistema.

Iniziamo dunque l'analisi identificando le tipologie di clienti che con più alta probabilità necessiteranno del nuovo sistema di gestione remoto.

Senza ombra di dubbio l'esigenza di un controllo centralizzato dei forni inizia ad emergere nelle installazioni di medie/grandi dimensioni, dove il numero di dispositivi da controllare supera le 5 unità, le quali molto probabilmente saranno distribuite sul territorio. Piccoli ristoratori, bar e pasticceri molto probabilmente non saranno interessati ad installare URC[3], in quanto i forni che utilizzano sono concentrati in 1/2 cucine e risultano direttamente accessibili al personale incaricato alla manutenzione.

Considerando quindi come probabili utilizzatori centri di medie/grandi dimensioni, procediamo con l'analizzare le strutture e le organizzazioni aziendali di riferimento e le esigenze più comuni.

2.2.1 Strutture e caratteristiche aziendali

La caratteristica essenziale delle aziende di ristorazione (o più in generale di produzione di alimenti) di una certa importanza sta nella loro dislocazione territoriale. A titolo di esempio possiamo pensare alla catena europea di supermercati discount LIDL, la quale opera in più di 22 paesi ed annovera in totale ben 16.000 centri di distribuzione. E' facile comprendere che in casi come questi ciò che è stato esposto al capitolo 1 si concretizza in delle esigenze reali.

Aziende di questo tipo, seppur di eccellenza, rappresentano comunque il modello di riferimento per la progettazione del sistema, che dovrà essere senza ombra di dubbio completamente scalabile e modulare. Per tutti gli altri casi occorrerà fare in modo che la complessità legata a sistemi di queste dimensioni non ne appesantisca l'utilizzo in aziende di dimensioni più contenute.

[3]URC sta per UNOX Remote Control; nel seguito del testo verrà utilizzato per indicare il sistema di controllo remoto, oggetto del presente documento.

La struttura aziendale dei grandi utilizzatori può essere schematizzata con il diagramma rappresentato in Figura 2.3. I diversi punti vendita (che installano i

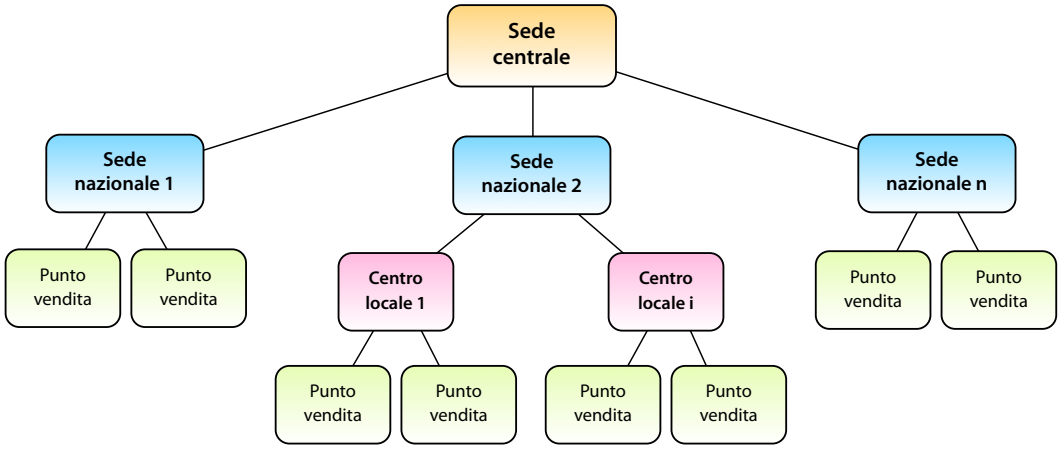

Figura 2.3: Rappresentazione dell'organizzazione aziendale dei clienti utilizzatori.

forni di cottura) sono collegati a delle strutture (a livello superiore) che coordinano le attività di quell'area. A loro volta anche queste strutture sono collegate a delle sedi nazionali, le quali ricevono le direttive dalla sede centrale. La struttura che si viene a creare prende quindi la forma di un albero, dove i vari nodi sono rappresentati dai vari centri distrettuali. Come evidenzia la Figura 2.3 è probabile che in alcune zone il numero dei sotto-livelli dell'albero sia maggiore in quanto l'organizzazione aziendale di quell'area più ossere più complessa.

In organizzazioni di questo calibro è probabile che vi sia la necessità di poter raccogliere i dati di funzionamento dei forni anche in più livelli, poiché le mansioni svolte da centri a livelli diversi possono essere differenti. Con riferimento alla Figura 2.3 è possibile che le sedi nazionali siano interessate a raccogliere dati di tipo statistico, per conoscere, ad esempio, il numero di programmi di cottura avviati giornalmente. Mentre i diversi centri locali, a livello inferiore, possono essere incaricati di occuparsi soltanto dell'assistenza. Pertanto l'architettura e le funzionalità del sistema di gestione remota dovranno permettere un accesso ai dati decentralizzato, che possibilmente a livello X dovrà però raggruppare i dati di tutti i forni presenti ai livelli inferiori.

Restando sempre in questo scenario è fondamentale sottolineare quanto sia importante riuscire ad analizzare i dati provenienti dai diversi forni in modo aggregato e strutturato. Per far ciò sarà opportuno prevedere che URC integri delle funzionalità di esportazione che all'occorrenza permettano di importare i dati raccolti in degli appositi tool di analisi.

Un altro aspetto importante è rappresentato dalle esigenze degli utilizzatori di ogni livello, che variano in base alle diverse mansioni svolte. E' lecito supporre che gli addetti alla manutenzione siano interessati ad accedere alle informazioni in modo più veloce e meno approfondito rispetto a chi dovrà occuparsi di generare dati statistici di funzionamento. Inoltre, è probabile immaginare che i primi abbiano bisogno di accedere ai dati anche in mobilità (con palmari o simili), mentre i secondi

lavoreranno la maggior parte del tempo stando seduti davanti ad un PC. Per queste ragioni è importante che la parte del sistema a diretto contatto con l'utente, ovvero l'interfaccia, sia studiata anche sulla base delle diverse esigenze operative. Inoltre occorre considerare che aziende che con la propria rete di vendita si estendono in diversi paesi hanno bisogno di un'interfaccia localizzata in più lingue e considerando che UNOX esporta i propri prodotti anche in paesi dell'estremo oriente, occorre prevedere il supporto anche di linguaggi orientali.

2.2.2 Infrastrutture hardware e software esistenti

Il nuovo sistema di controllo remoto avrà bisogno di infrastruttura informatica per la raccolta, l'archiviazione e la gestione dei dati. Per questo motivo risulta importante studiare la più probabile composizione dei sistemi informativi della clientela, al fine di progettare un sistema la cui messa in funzione da parte dell'utilizzatore non comporti troppe spese.

Naturalmente sarà difficile pensare di realizzare un sistema di questa portata che sia in grado di adattarsi (ed eventualmente integrarsi) a tutte le tipologie di sistemi esistenti, ma in questa fase qualche accorgimento potrà sicuramente tornare utile.

Considerata la natura e le dimensioni della clientela a cui è rivolto il sistema è lecito pensare che tali aziende dispongano già di una infrastruttura informatica di base, con uno o più PC e, nei casi migliori, di uno o più server per la gestione dei dati. Probabilmente nelle aziende più strutturate è possibile che ogni sede direzionale distribuita sul territorio sia dotata di uno o più apparai server.

Considerando che la maggior parte dei client e dei server aziendali installa sistemi operativi di casa Microsoft, è altamente probabile che nella stragrande maggioranza dei casi il sistema dovrà essere integrato in tali piattaforme.

In ogni caso bisogna prestare attenzione anche ai casi differenti (sistemi Unix-like), soprattutto per quanto riguarda le unità client. Inoltre, in base a quanto esposto alla sezione precedente, la 2.2.1, è probabile che una parte degli utilizzatori, soprattutto quelli addetti alla manutenzione, abbia bisogno di utilizzare il sistema in mobilità, analizando i dati attraverso dei dispositivi portatili come palmari o tablet. Pertanto, considerando tali esigenze, sarebbe opportuno assicurare che la parte del sistema a stretto contatto con l'utente sia compatibile con tutta questa categoria di prodotti.

2.2.3 Esigenze della clientela

Durante le fasi di avvio dell'attività progettuale si è svolta in UNOX una riunione a cui hanno partecipato, oltre ai responsabili del progetto, gli incaricati alla gestione dei clienti direzionali UNOX. I clienti direzionali sono tutte quelle aziende di medio/-grandi dimensioni per le quali UNOX applica un trattamento privilegiato, attraverso la vendita diretta dei propri prodotti senza far uso, cioè, di agenti intermediari.

In tale occasione tutte le parti presenti si sono confrontate al fine di raccogliere le esigenze della clientela e fare in modo che queste vengano soddisfatte con l'introduzione del nuovo sistema.

I punti salienti di ciò che è emerso sono i seguenti:

- *I clienti hanno bisogno di aggiornare i programmi di cottura delle maschere comandi dei forni*
  In grandi organizzazioni le pietanze vendute sono le stesse in tutti i punti vendita, o perlomeno sono le stesse a livello nazionale/locale. Pertanto la direzione generale (o locale) ha bisogno di poter programmare tutti i forni allo stesso modo, ovvero fare in modo che i programmi di cottura presenti in ogni forno siano gli stessi.

- *In modo simile ai programmi di cottura, il sistema deve permettere l'aggiornamento di alcuni parametri di funzionamento dei forni*
  La maschera comandi di ogni forno, oltre a permettere agli utenti di salvare i propri programmi di cottura, mette a disposizione un menu nascosto attraverso il quale è possibile agire su alcuni parametri di funzionamento[4].

- *Il sistema dovrà permettere l'esportazione di tutti i dati di log HACCP*
  HACCP è l'acronimo di Hazard Analysis and Critical Control Points e in sostanza consiste in un sistema di autocontrollo che ogni operatore nel settore della produzione di alimenti deve mettere in atto al fine di valutare e stimare pericoli e rischi, oltre che stabilire misure di controllo per prevenire l'insorgere di problemi igienici e sanitari. I dati che interessano sono le registrazioni a cadenze regolari di tutti i parametri di cottura di un cibo, quali temperatura camera, temperatura cuore, grado di umidità, ecc, più altri parametri per registrare l'avvio delle procedure di pulizia dei forni (sistema Rotor Klean). Malfunzionamenti del forno possono compromettere la buona riuscita dei programmi di cottura e tali irregolarità possono essere rilevate attraverso l'analisi dei log di cottura.

- *Il sistema dovrà permettere il recupero dei tabulati relativi ai programmi avviati*
  Al fine di raccogliere e analizzare i dati sulle vendite, anche in tempo reale, gli utilizzatori sono interessati a recuperare l'elenco dei programmi di cottura avviati.

- *I clienti sono interessati a rilevare eventuali warning e allarmi*
  L'elettronica interna ai forni UNOX permette di recuperare dati per rilevare il corretto funzionamento di tutti i dispositivi (resistenze, ventole, ecc) sia durante le operazioni di cottura sia durante l'avvio di programmi di autodiagnosi. Pertanto al fine di attuare un programma di manutenzione ottimale, l'analisi da remoto di tali informazioni risulta indispensabile.

Un'altra importante funzionalità che può essere introdotta con il sistema (che non rappresenta un'esigenza diretta degli utilizzatori) sta nella possibilità di aggiornare i firmware interni alle schede di controllo dei prodotti UNOX. Oggigiorno la maggior parte dei dispositivi che integrano sistemi di accesso remoto offrono tale funzionalità. Essa permette di risolvere eventuali problemi di funzionamento, oltre che aggiungere nuove funzionalità ai prodotti esistenti. Attualmente UNOX, attraverso un'apposita

[4] solitamente il personale che si occupa di tali aspetti è quello addetto all'installazione/manutenzione degli impianti.

interfaccia di controllo chiamata OVEX.Net (che verrà presentata nel seguito del testo), permette di eseguire tale procedura copiando il nuovo firmware in un supporto di memoria USB. Ma la possibilità di svolgere tale attività anche da remoto rappresenta senza dubbio una grande opportunità.

2.3 Le specifiche del sistema

In questa sezione del testo verranno riassunti brevemente tutti gli aspetti legati agli argomenti trattati finora, al fine di delineare la struttura hardware e software di URC.

Partendo dalla struttura interna ai forni UNOX possiamo stabilire che il nuovo sistema di gestione remota dovrà:

- integrarsi nelle diverse unità forno come optional; dovrà quindi poter essere installato separatamente dal resto dell'elettronica di bordo;
- dialogare con le altre unità elettroniche di controllo via Modbus, al fine di raccogliere e importare tutti i dati necessari;
- comportare il minor numero possibile di modifiche lato software ai firmware delle unità di controllo esistenti e, possibilmente, nessuna modifica all'hardware.

Tipo	Descrizione
Importazione	Aggiornamento firmware
Importazione/Esportazione	Programmi di cottura
Importazione/Esportazione	Parametri di funzionamento
Esportazione	Log HACCP
Esportazione	Log programmi di cottura avviati
Esportazione	Warning e allarmi
Esportazione	Risultati dei programmi di autodiagnosi

Tabella 2.2: elenco delle routine di importazione/esportazione dati

Per quanto riguarda gli aspetti legati agli utilizzatori finali il sistema dovrà essere:

- scalabile, per poter essere configurato secondo l'organizzazione interna alle aziende (struttura ad albero);
- portabile, al fine di permettere agli operatori un utilizzo in mobilità;
- multipiattaforma, per abbattere i costi di installazione e messa in funzione su infrastrutture informatiche differenti;
- multilingua, per poter essere utilizzato senza difficoltà dagli utilizzatori di tutto il mondo;
- performante, in grado di gestire un grande quantitativo di dati.

Le funzionalità messe a disposizione dal sistema vengono riportate in Tabella 2.2. Le diverse funzionalità nel seguito del testo verranno definite come *Routine di importazione/esportazione dati*, per fare riferimento allo scambio fisico di dati fra forni UNOX ed il mondo esterno.

Capitolo 3

Architettura del sistema URC

Dopo un'attenta analisi di tutte le problematiche connesse al progetto e dopo aver definito le specifiche del sistema, passiamo alla fase di progettazione, la quale inizia con una serie di valutazioni sulle possibili soluzioni implementative.

Per iniziare occorre definire in quale modo estrarre i dati dalle logiche integrate dei forni, ovvero come progettare un dispositivo che possa essere interfacciato al bus Modbus di sistema; in seguito bisogna stabilire quali canali e quali protocolli di comunicazione utilizzare per veicolare le informazioni dai forni alle diverse sedi dei clienti utilizzatori.

I dati in uscita dai forni dovranno poi essere ricevuti in modo opportuno, per l'archiviazione, la consultazione e l'analisi. A tal scopo occorre progettare un sistema informativo capace di adempiere a tutte le esigenze della clientela discusse alla sezione 2.2.

3.1 Il nuovo bridge elettronico

Considerando che i forni attualmente prodotti da UNOX sono già provvisti di ottime schede di controllo, l'elettronica preposta al nuovo scopo, cioè quello della trasmissione dei dati verso degli host remoti dovrà essere realizzata appositamente.

In sostanza si dovrà pensare di realizzare una nuova scheda elettronica provvista di tutto l'hardware necessario per eseguire le seguenti attività:

- interfacciamento al bus di sistema per il dialogo con le altre interfacce di controllo;
- salvataggio dei dati ricevuti (o da inviare) in memoria;
- ricezione e trasmissione dei dati da e verso i sistemi informativi dei clienti utilizzatori.

Prima di affrontare nelle specifico le diverse problematiche è bene fare una precisazione.

UNOX già da qualche anno dispone di un sistema (o meglio di un optional) simile, che permette di aggiornare i programmi di cottura e il firmware interni ai forni.

Questo prodotto prende il nome di OVEX.Net e consente di svolgere le operazioni di aggiornamento attraverso una memoria USB. Il sistema è dotato di un *bridge* (scheda elettronica) e di un software per PC che permette di editare e salvare i programmi di cottura in maniera completamente visuale e di riversarli in una memoria USB, la quale viene utilizzata per lo scambio dei dati fra PC e forno. Una volta inserita la memoria nell'apposito bridge OVEX.Net a bordo del forno, attraverso la maschera comandi del forno stesso l'utente può selezionare i diversi programmi di cottura salvati in memoria e caricarli nella scheda di controllo primaria del forno. In questo modo l'editazione dei programmi di cottura e la programmazione di più forni può essere eseguita in poco tempo. La stessa procedura può essere seguita per effettuare l'aggiornamento del firmware delle diverse schede elettroniche interne al forno.

OVEX.Net, quindi, dispone già dell'hardware necessario a svolgere le attività elencate in precedenza, con la sola differenza che il prelievo e il caricamento dei dati non può essere svolto da remoto, ma solo fisicamente attraverso l'utilizzo di una memoria USB.

Partendo da tale presupposto, per la progettazione del nuovo hardware è stato deciso di sviluppare una nuova versione della scheda di controllo OVEX.Net con l'aggiunta di un'interfaccia dedicata alla ricetrasmissione dei dati.

3.1.1 L'interfaccia fisica di comunicazione

Come è stato osservato più volte, l'installazione standard del nuovo sistema prevede che le diverse sedi operative dei clienti, ognuna delle quali può installare uno o più forni, possano trovarsi a notevole distanza (anche su stati differenti). Pertanto, se tutti i dati devono poter essere raccolti in sede centrale (o nei vari centri locali), occorrerà inviarli sfruttando una rete di comunicazione esistente, la cui estensione non rappresenti un limite alla propagazione delle informazioni.

In tale scenario appare quindi chiaro che l'unica rete che consente di ottenere tali risultati, garantendo inoltre alte velocità di trasmissione dei dati, è la rete Internet. Oggigiorno Internet, grazie alle decine di protocolli di trasporto differenti, si estende praticamente a livello globale, su mezzi fisici e standard completamente differenti: reti PSTN, ISDN, xDSL, GPRS, UMTS, fibre ottiche, link radio ad alta frequenza, onde convogliate, ecc. Pertanto integrare in URC un'interfaccia che permetta la trasmissione dei dati attraverso Internet rappresenta, oltre che una grande occasione, una necessità.

Le interfacce fisiche che possono essere utilizzate a tale scopo sono molteplici, ma negli ultimi tempi nell'ambito dell'elettronica embedded l'integrazione in molti microcontrollori di interfacce Ethernet ha fatto sì che tale standard possa essere implementato contenendo tempi di sviluppo e costi di produzione. A favorire lo sviluppo di applicazioni di questo tipo contribuisce l'accesso praticamente capillare a reti Ethernet private (LAN); grazie alla diffusione delle reti xDSL e di modem router con interfaccia Ethernet lo standard si è di fatto imposto a livello mondiale, sia nel settore privato che dell'impresa.

Il nuovo bridge prodotto da UNOX sarà quindi equipaggiato con interfaccia Ethernet la quale, come vedremo in seguito, sarà integrata all'interno del micro-

controllore di bordo.

3.1.2 I protocolli di trasporto dei dati

Quando si parla di Internet risulta immediato parlare di *suite di protocolli TCP/IP*, la quale prende il nome dai due principali protocolli di incapsulamento dei dati: il TCP e l'IP.

In realtà il trasporto dei dati su Internet fa uso di svariati protocolli, i quali nell'implementazione software classica assumono la struttura dettata dallo standard OSI a 7 livelli, rappresentato in Figura 3.1.

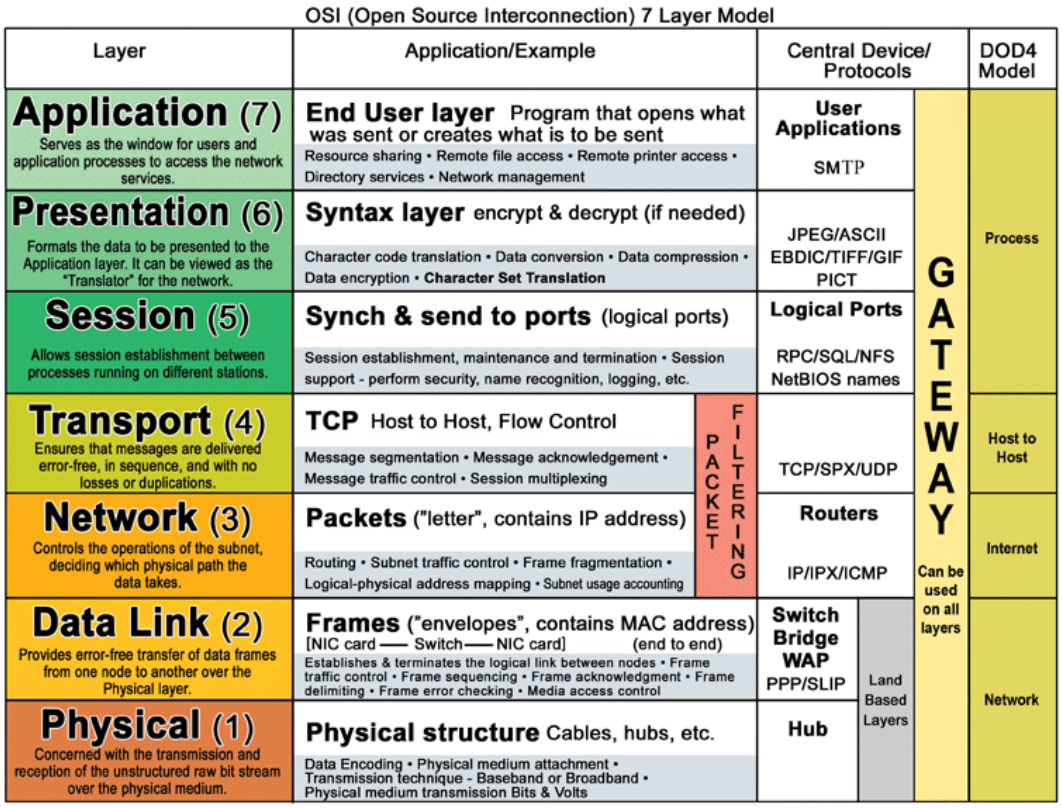

Figura 3.1: Pila protocollare dello standard OSI a 7 livelli.

Quindi per implementare un sistema che sia in grado di interfacciarsi alla rete Internet occorre prevedere, oltre alla parte di interfaccia fisica, l'implementazione software dei protocolli necessari alla comunicazione. Nel campo dell'elettronica embedded, grazie alla diffusione di sistemi dotati di interfacciamento ad Internet, sono state sviluppate moltissime differenti implementazioni della pila protocollare TCP/IP. Tali pacchetti software prendono il nome di *Stack TCP/IP*, e rispetto alle versioni diffuse nei più moderni SO per PC, si caratterizzano per essere molto più compatti in quanto forniscono solo le funzionalità essenziali. Di conseguenza si possono trovare implementazioni dei protocolli TCP/IP che in linguaggio macchina

occupano poche decine di kByte e che permettono di gestire l'esigua memoria a bordo dei microcontrollori in maniera del tutto efficiente.

Oggigiorno la moltitudine di Stack TCP/IP presenti in commercio consente di realizzare un sistema con accesso ad Internet davvero in poco tempo e a costo zero: molte versioni della pila protocollare vengono infatti distribuite sotto licenza GNU GPL (o simili), molte volte dalle stesse case di produzione di circuiti integrati.

3.2 L'applicativo di raccolta dati

Oltre a dover ricevere i dati in modo corretto bisogna fare in modo di poterli archiviare in modo permanente, in modo da consentire analisi statistiche con relative stampe o esportazioni.

La soluzione senza ombra di dubbio più appropriata consiste nel realizzare un'applicazione per computer che, una volta avviata, si metta in connessione con i bridge di uno o più forni, al fine di recuperare le informazioni che dovranno poi essere riversate in un'apposita base di dati.

Occorre pertanto trovare il modo più appropriato per realizzare sia l'applicazione, sia il database di appoggio, definendo le diverse possibilità di configurazione in base a quanto osservato nell'analisi del contesto.

3.2.1 Il paradigma client-server

La rete Internet e le sue applicazioni più comuni si basano sul *paradigma client-server*, secondo il quale lo scambio di dati fra due nodi inizia con un'interrogazione del nodo client all'unità server. Una volta che il client ha stabilito con successo la connessione con il server, inviandogli unitamente la sua richiesta, il server risponde inviando i dati richiesti. Nella rete Internet i server vengono contattati utilizzando il loro indirizzo IP (o in alternativa attraverso un URL sfruttando il servizio DNS); solitamente oltre all'indirizzo (a meno che non sia predisposta una struttura hardware e/o software che esegua tale procedura in automatico), occorre specificare anche la porta TCP a cui connettersi.

Ipotizzando di adottare tale soluzione anche in URC, occorre stabilire i ruoli svolti da ogni parte, e in particolare da bridge Ethernet e applicativo software di raccolta dati. La tabella 3.1 descrive le due diverse possibilità, mettendo in evidenza pregi e difetti di ogni soluzione sotto il profilo pratico.

Gli svantaggi di ogni soluzione sono stati largamente discussi in UNOX durante le fasi di progettazione e le considerazioni emerse durante l'analisi sono le seguenti:

- è probabile che utilizzatori di medie/grandi dimensioni con più punti vendita e un certo grado di informatizzazione siano dotati di un responsabile informatico, o perlomeno che si appoggino a tecnici esterni qualificati. Pertanto le prime due problematiche elencate nella prima ipotesi possono essere risolte attuando una corretta configurazione dei dispositivi di protezione (router/firewall);

Server	Client	Vantaggi	Svantaggi
Forno	Software raccolta dati	Prelievo dati su richiesta: l'applicativo può contattare il forno solo quando ha effettivo interesse a recuperare i dati	Se forno e applicativo software non sono nella stessa rete LAN eventuali firewall possono impedire la comunicazione. In punti vendita con più forni vanno messe in atto politiche di NAT per dialogare con forni su porte TCP differenti. Ogni punto vendita deve essere dotato di un IP di rete statico. Vanno gestite le richieste simultanee verso il forno per evitare di saturare la limitata capacità computazionale del server.
Software raccolta dati	Forno	I firewall e i dispositivi di protezione della rete vengono by-passati	Non è possibile interrogare il forno a piacere in quanto l'avvio delle comunicazioni avviene in senso inverso. Alcune tipologie di trasmissione dei dati devono essere temporizzate, pertanto i dati non sono aggiornati in tempo reale.

Tabella 3.1: differenti configurazioni del modello client-server in URC.

- ancora per quanto concerne il primo caso, anche la terza problematica può essere risolta in modo semplice, ad esempio appoggiandosi ad un servizio di dynamic DNS, per associare un IP dinamico ad un indirizzo DNS;
- il quarto svantaggio della prima soluzione può essere risolto attuando nell'applicativo software una corretta politica di invio delle richieste al server;
- gli svanataggi della seconda soluzione non possono essere evitati, a meno che non si utilizzino delle connessioni TCP permanenti. Ma riprendendo il caso LIDL, con 16.000 punti vendita, è possibile che in tale scenario un applicativo server debba gestire centinaia di connessioni simultanee, con conseguente degrado delle performance globali e spreco di banda.

Dopo questa analisi è stato scelto di optare per la prima soluzione, integrando nel

firmware del bridge Ethernet un'applicazione ad alto livello per la realizzazione del server TCP e scrivendo un applicativo software provvisto di un modulo client in grado di stabilire le connessioni per il recuero dei dati.

3.2.2 La struttura ad albero

Un'esigenza già analizzata alla sezione 2.2.1 prevede che le informazioni derivanti da un certo sotto-gruppo di forni debbano poter essere consultate da tutte le sedi di livello superiore. Per chiarire tale concetto si può pensare al seguente esempio: facendo riferimento al caso LIDL, consideriamo tutti i forni installati nel nord Italia. I dati in uscita da questi forni saranno raccolti dal centro direzionale del nord Italia per tutte le attività di assistenza, come la verifica dello stato dei forni e l'aggiornamento dei programmi di cottura. Ma supponendo che i dati statistici vengano trattati a più livelli, è possibile che le informazioni riguardanti i log HACCP e i log delle cotture avviate debbano essere consultati dalla divisione nazionale ed centrale (con sede in Germania). Da tale esempio si comprende che tutti i dati (o una determinata categoria) devono poter essere consultati da ogni livello di gerarchia aziendale, i dati cioè devono poter essere raccolti da sistemi informativi indipendenti, quali possono essere ad esempio il sistema informativo della sede Italiana e quello della sede centrale Tedesca.

Si è quindi pensato di realizzare un appplicativo software di raccolta dati altamente flessibile e che permetta diverse tipologie di configurazione.

L'idea che sta alla base del sistema è quella di realizzare un applicativo che nelle installazioni più complesse possa essere "inserito" in una struttura ad albero, nella quale ogni nodo (ovvero ogni applicativo) sia interconnesso ad altri nodi secondo la gerarchia aziendale dell'utilizzatore.

Pertanto in installazioni che contano pochi punti vendita e un solo centro di raccolta dati saremo in presenza di un albero a due livelli, cioè con un unico nodo radice (l'applicativo di raccolta dati) ed X foglie, tante quante sono i forni installati nei vari punti vendita (X = numero forni).

Nel caso di una grande azienda invece, saremo in presenza di una struttura ad albero di N livelli ed X foglie, dove X è sempre pari al numero di forni installati, mentre N dipende dal grado di verticalizzazione dell'organizzazione aziendale. In ogni caso il nodo al vertice di un determinato sotto-albero (composto da più nodi interni ed Y foglie, con $Y < X$) avrà a disposizione i dati di tutti i forni, dati che gli sono stati comunicati dai nodi ai livelli inferiori, i quali sono direttamente connessi ai forni di cottura.

La più generica configurazione può essere dunque rappresentata come in Figura 3.2.

Va evidenziato che la caratteristica principale di questa soluzione prevede che ogni nodo interno (cioè ogni software di raccolta dati) sia identico. In pratica significa che l'applicativo software realizzato è universale e può essere installato a qualsiasi livello. Ciò che cambia sarà la sola configurazione di ogni installazione, anche se come vedremo in seguito questa sarà limitata a pochi parametri.

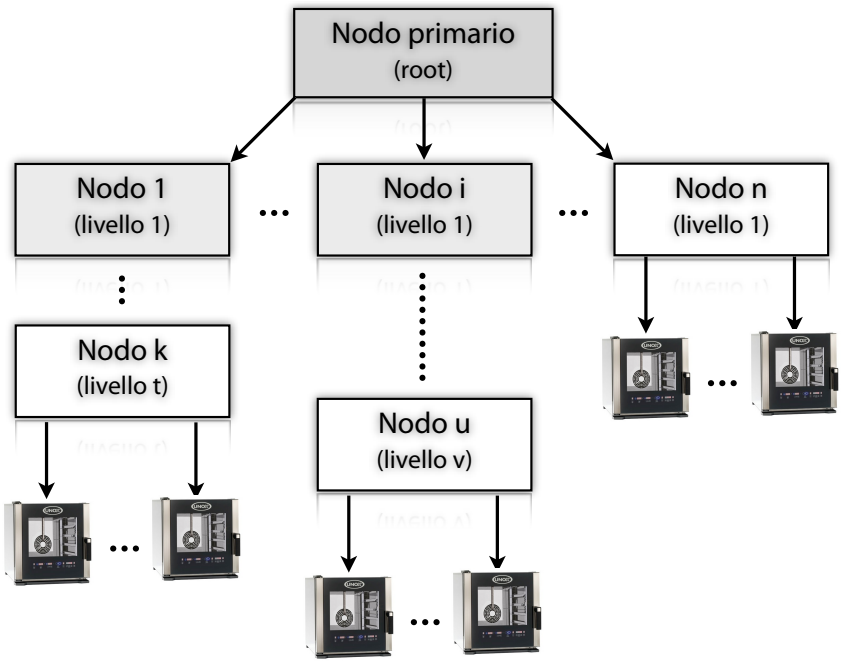

Figura 3.2: URC nella sua configurazione ad albero.

Nel seguito del testo ogni nodo dell'albero che installa il software di raccolta dati verrà chiamato *nodo server*, o più semplicemente *server,* mentre con il temrine *bridge* si farà riferimento al sistema a bordo del forno (scheda con interfaccia Ethernet e relativo firmware con stack TCP/IP). Questa convenzione, che può lasciar pensare un abuso di linguaggio (con il termine *server* finora si è indicato l'applicativo interno al bridge dei forni), in realtà è lecita e permette di descrivere con maggior precisione le diverse parti del sistema. Infatti, è probabile che ogni applicativo software di raccolta dati venga installato all'interno dei server aziendali dei clienti utilizzatori, ovvero in computer ad alte performance e dotati di memoria secondaria ridondata (o in alternativa di sistemi di backup dei dati). Per cui da ora in avanti si utilizzerà il termine *server* per indicare tali dispositivi, e, per estensione, l'applicativo UNOX installato.

Caratteristiche della struttura del sistema Sempre in riferimento alla Figura 3.2 verrà ora riportato un elenco di caratteristiche del sistema. Ogni voce in elenco è stata definita durante le fasi di progettazione, e deriva da una serie di considerazioni che sono emerse durante le riunioni dello staff coinvolto nel progetto:

1. il sistema di controllo remoto assume complessivamente una struttura ad albero;
2. i nodi interni dell'albero sono rappresentati dalle unità server;
3. le foglie dell'albero sono rappresentate dalle unità forno (forno singolo o colonna di forni);

4. ogni server installa la stessa applicazione di raccolta e invio dati;
5. ogni server viene configurato per "conoscere" solo la lista dei propri figli (ad es. il server a livello k conosce solo i server del livello $k+1$);
6. è possibile che server che non stanno su livelli adiacenti non siano visibili in rete (ad es. il server primario - radice - con il server k di livello t);
7. i nodi (interni o foglie) non "conoscono" e quindi non possono contattare il proprio padre;
8. ogni server può immagazzinare tutti i dati di funzionamento dei server a livello inferiore in un apposito database;
9. il nodo radice (server primario) immagazzina i dati di tutti gli altri nodi, quindi di conseguenza i dati di tutti i forni;
10. ogni nodo dell'albero deve essere configurato per poter essere identificato con un nome (stringa di testo) univoco;
11. il nome attribuito alle foglie dell'albero (forni o colonne di forni) risiede all'interno del relativo server padre.

La maggior parte delle caratteristiche elencate dovrebbe risultare chiara, in quanto fanno riferimento ad aspetti già discussi, mentre altre caratteristiche risulteranno chiare in seguito.

Le uniche caratteristiche che meritano un approfondimento anche in questa parte del testo sono la 3 e la 11. In questi punti, infatti, si sottolinea che le foglie dell'albero sono costituite da singoli forni o da singole colonne di forni. Questo aspetto deriva dal fatto che, come è già stato osservato alla sezione 2.1.1, i forni installati in un'unica colonna sono connessi allo stesso bus Modbus; pertanto ogni colonna disporrà di un solo bridge, il quale potrà comunque dialogare con le logiche di controllo di tutti i forni della colonna.

3.2.3 Il servizio di sistema e la base di dati

L'applicativo di raccolta dati installato presso i server dei clienti dovrà integrare una serie di routine software sempre attive per monitorare lo stato dei forni e raccogliere tutti i dati di interesse.

In un'ottica di lavoro in ambienti server (la quale non preclude comunque l'utilizzo in altri ambienti - es. installazione in un comune PC) è opportuno pensare di implementare il software realizzando un apposito servizio di sistema, il quale viene avviato automaticamente all'avvio del sistema operativo e termina con l'arresto dello stesso.

A questo punto è opportuno fare una piccola precisazione: finora si è detto che tale applicativo dovrà integrare un client TCP capace di comunicare con l'applicativo server presente nel bridge Ethernet. Ma considerando la struttura ad albero rappresentata in Figura 3.2, è necessario che ogni nodo dell'albero (ovvero ogni servizio di

sistema) sia in grado di dialogare anche con il proprio padre, al fine di trasferirgli le informazioni di cui è in possesso. Pertanto risulta ovvio che il servizio di sistema, oltre ad un modulo client, debba integrare un secondo modulo di tipo server, pronto a rispondere alle richieste del padre di livello superiore.

Come è già stato detto in precedenza, oltre alle funzionalità preposte alla comunicazione, il servizio di sistema sarà impegnato in un continuo salvataggio e reperimento di dati. Per questo motivo l'applicativo dovrà far uso di una base di dati di appoggio, necessaria per effettuare le operazioni di scrittura, lettura e modifica dei dati in modo efficiente.

3.2.4 Il formato di scambio dati: l'XML

Arrivati a questo punto è necessario stabilire in quale modo i diversi nodi server possono scambiarsi le informazioni. Esse sono di nature e forme differenti (es. testi, stringhe di byte, file HEX di aggiornamento del firmware, ecc). La stesso discorso, però, può essere fatto anche per le comunicazioni fra nodi server e forni.

In sintesi risulta quindi necessario trovare un sistema per incapsulare in modo intelligente dati di natura diversa, garantendo, comunque, che tale trasformazione possa essere svolta in poco tempo e attraverso l'utilizzo di poche risorse hardware. Questo perché, una volta scelto il formato di incapsulamento dei dati, anche il firmware interno al bridge dovrà essere in grado di interpretare e trasmettere dati appartenenti al linguaggio facendo uso di risorse limitate.

Per trasmettere le informazioni fra i diversi nodi (sia server, ovvero servizi di sistema, sia forni) si è scelto di utilizzare lo standard XML. Si tratta di un metalinguaggio nato alla fine degli anni '90, che oggigiorno viene utilizzato in moltissimi sistemi proprio nelle procedure di scambio dati (es. importazione/esportazione, in ambito Web nelle applicazioni che fanno uso di AJAX, ecc). Essendo un metalinguaggio, l'XML consente di definire nuovi linguaggi a piacere, adattando le regole di ogni nuovo linguaggio alle esigenze specifiche.

Nel caso di URC, come vedremo in seguito, ogni tipologia di comunicazione farà uso di XML differenti: ad esempio l'invio dei programmi di cottura utilizzerà un XML completamente differente da quello usato per il prelievo dei log HACCP.

3.3 L'interfaccia utente

Lo studio e la progettazione dell'interfaccia utente è un'attività che molto spesso viene sottovalutata perché nella realizzazione di nuovi dipositivi le aziende tendono ad investire molte risorse nello sviluppo delle routine di sistema, senza curare, o curando in minima parte, i livelli dell'applicazione a diretto contatto con l'utente. L'errore che di solito viene commesso è quello di considerare l'utente un "esperto" del settore e credere che, in un modo o nell'altro, riuscirà a compiere il proprio lavoro senza particolari problemi. Va però considerato che, soprattutto in sistemi di tipo software, la maggior parte degli utenti non ha una completa dimestichezza con l'uso del computer e di conseguenza un'interfaccia utente poco intuitiva può appesantire l'esperienza di utilizzo dell'intero sistema.

UNOX ha dunque espresso la volontà di realizzare per URC un'interfaccia di alto livello, investendo notevoli risorse sia per la progettazione che per lo sviluppo, cercando inoltre di coprire le esigenze viste nei capitoli precedenti.

Prima di proseguire con l'analisi occorre evidenziare il seguente aspetto: come è stato visto l'applicativo dedicato alla raccolta dei dati è un servizio di sistema ed in quanto tale non sarà provvisto di interfaccia utente. Per tale motivo risulta necessario mettere a punto un applicativo di interfaccia utente dedicato, che chiaramente dovrà essere in grado di comunicare con il servizio di sistema per lo scambio dei dati.

Il canale di comunicazione per lo scambio dati può essere realizzato sostanzialmente in due modi distinti:

- integrando nel servizio di sistema un modulo in grado di comunicare con agenti esterni, ad esempio attraverso un socket TCP;
- utilizzando il database di appoggio come area comune per la scrittura e la lettura dei dati.

Come è già stato fatto per gli altri moduli del sistema URC, prima di stabilire quale può essere la soluzione più idonea, risulta opportuno considerare le specifiche che dovranno essere soddisfatte. Nel caso dell'interfaccia utente bisognerà garantire: portabilità, compatibilità (sistema multipiattaforma) e traduzioni multilingua.

In termini di portabilità occorre prevedere che il servizio possa essere utilizzato in mobilità. Per questo, oltre a garantirne la compatibilità con i classici SO per computer portatili, è opportuno fare in modo che l'applicativo possa essere eseguito su dispositivi mobile di ultima generazione, come palmari e tablet.

In tale scenario è proprio la compatibilità a giocare un ruolo chiave, in quanto il modulo di interfaccia dovrà funzionare anche in SO appartenenti alla famiglia mobile, come ad esempio iOS di Apple e Android di Google.

La localizzazione in multilingua fa invece parte delle funzionalità richieste dal sistema, pertanto non rappresenta un vincolo.

Considerando questi aspetti si è quindi pensato di svincolare l'interfaccia utente da qualsiasi tipo di compatibilità, in modo da garantirne l'effettivo utilizzo in ambienti e applicazioni differenti. A tal proposito l'idea è stata quella di implementare un'interfaccia utente Web-based, seguendo la tendenza attuale che vede le applicazioni Web sempre più al centro della scena.

Rimane ora da definire quale tipo di canale di comunicazione sia opportuno utilizzare per lo scambio dati fra applicativo di interfaccia e servizio di sistema. In tal senso la scelta di UNOX è stata quella di minimizzare gli sforzi e di sfruttare il canale esistente, facendo cioè in modo che l'applicativo di interfaccia Web-based acceda direttamente ai dati presenti nel database per recuperare i dati richiesti dall'utente.

Tale scelta presenta comunque due conseguenze fondamentali, che dovranno essere considerate per lo sviluppo attuale e futuro del sistema:

- i dati presenti nel database "condiviso" devono poter essere letti, interpretati e manipolati da entrambi i moduli. Questo implica che, ad esempio, eventuali codici di riferimento per i dati dovranno essere conosciuti sia dal servizio di sistema sia dall'applicativo di interfaccia (questo aspetto risulterà più chiaro in seguito);

- futuri aggiornamenti sulla struttura tabellare della base di dati prevedono il conseguente aggiornamento di entrambi i moduli.

Le applicazioni Web-based Con l'espressione *applicazione Web-based* si identificano solitamente tutti i moduli software realizzati con linguaggi utilizzati per la costruzione di siti Web e che per tale motivo sono soggetti ai criteri di sviluppo utilizzati in tale settore.

L'aspetto fondamentale è comunque il seguente: nelle applicazioni Web-based l'utente utilizza il sistema direttamente attraverso un comune browser Web, proprio come se stesse navigando su Internet. Per questo motivo le applicazioni Web-based possono essere paragonate ad un sito, nel quale l'interfaccia viene impaginata e gestita attraverso linguaggi lato client, (che vengono cioè interpretati direttamente dal browser) mentre le funzionalità di gestione dei dati vengono gestite dal motore lato server. Come è noto, le applicazioni web-based, per poter funzionare, debbono essere installate ed eseguite in un server Web, come ad esempio Apache o IIS di Microsoft (la scelta del server dipende fondamentalmente dal linguaggio di programmazione usato). Applicato al caso URC significa che i vari centri di controllo che installano il servizio di sistema dovranno installare anche un Web server dedicato all'esecuzione dell'applicazione di interfaccia utente.

3.4 Schema a blocchi del sistema URC

Dopo aver analizzato i diversi moduli che compongono il sistema URC, riassumiamo quanto detto servendoci dello schema a blocchi riportato in Figura 3.3.

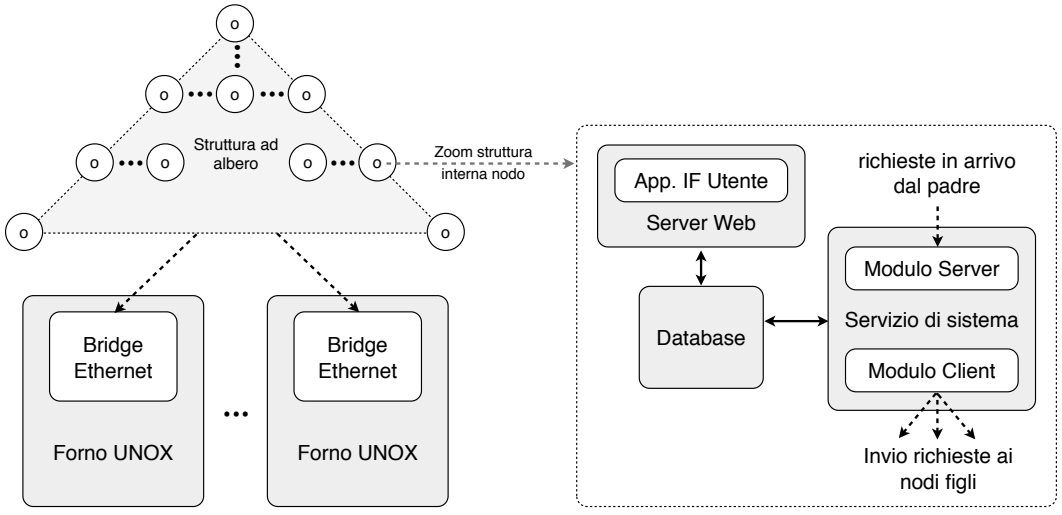

Figura 3.3: Schema a blocchi della struttura del sistema.

Come è visibile dall'immagine, la struttura del sistema si compone di diverse parti. Nell'insieme potrebbe risultare abbastanza complessa e quindi di difficile integrazione nei sistemi informativi degli utilizzatori. Ma, come vedremo in seguito

tutti i vari moduli potranno essere integrati all'interno di un unico pacchetto di installazione, che semplifica la messa in funzione del sistema a tal punto che l'intera collezione di software agli occhi degli utenti apparirà come un unico applicativo.

Nella parte inferiore dello schema a blocchi troviamo i forni UNOX installati nei diversi punti vendita. Ogni forno è dotato del nuovo bridge con interfaccia Ethernet, che sarà opportunamente collegato alla rete LAN del punto vendita.

I diversi nodi di monitoraggio sono collegati ai forni tramite Internet, o più in generale tramite una qualsiasi tipologia di rete che consenta la trasmissione di dati tramite IP (reti VPN, reti wireless punto-punto, ecc).

Ogni nodo server è costituito dai diversi blocchi rappresentati a destra dell'immagine, ovvero:

- il servizio di sistema con i moduli client e server. Il modulo client permette di inviare le richieste ai nodi di livello inferiore dell'albero, mentre il modulo server consente di accettare le richieste in arrivo dal nodo padre di livello superiore;
- il database per la memorizzazione di tutti i dati di interesse. Il database viene utilizzato anche come canale di comunicazione fra servizio di sistema e modulo di interfaccia utente;
- interfaccia utente che è composta da un applicativo web-based in esecuzione su un Web server.

3.5 Installazione di URC in UNOX come servizio al cliente

La struttura del sistema appena analizzata permette uno sviluppo ulteriore, a cui non si è ancora fatto alcun cenno.

Supponendo che UNOX intenda ampliare l'offerta del proprio programma di assistenza al cliente, sarà possibile installare in sede UNOX in server di raccolta dati direttamente collegato ai server delle sedi centrali dei clienti. In questo modo UNOX potrebbe raccogliere i dati di funzionamento dei forni installati presso i punti vendita di tutti i clienti e monitorarne lo stato, offrendo così un servizio di assistenza avanzato.

Questo permetterebbe ad UNOX di raccogliere anche dati statistici utili per identificare le parti (o i prodotti) più soggetti a guasti, o per rilevare il grado di utilizzo di ogni famiglia di prodotti.

Naturalmente per fare in modo che il server in sede UNOX possa recuperare i dati del sistema di gestione di un determinato cliente, occorre che il server centrale del cliente stesso sia accessibile dall'esterno. In pratica significa che i dispositivi di protezione della rete del cliente (firewall hardware e software) non dovranno bloccare connessioni provenienti dall'esterno e dirette verso la porta TCP di accesso al modulo server dell'applicativo di raccolta dati.

Capitolo 4

Il servizio di sistema

Dopo aver eseguito l'analisi critica dell'ambiente di utilizzo e dopo aver definito la struttura hardware e software del sistema, il presente capitolo è dedicato alla presentazione degli aspetti legati all'implementazione del software di raccolta dati.

Il capitolo inizia con l'introduzione all'ambiente di sviluppo scelto, per poi passare a presentare la struttura base di un servizio di sistema.

La seconda sezione è invece dedicata alla progettazione della base di dati di appoggio, che, come è già stato detto, verrà utilizzata anche dall'applicativo di interfaccia utente come area comune per il passaggio dei dati.

La parte conclusiva del capitolo è invece dedicata all'analisi del codice del programma; dapprima verrà descritta la struttura implementativa dell'intera applicazione, poi verranno analizzate le specifiche dei diversi XML utilizzati per lo scambio dati fra servizio padre e servizio figlio (quindi fra due livelli dell'albero) ed infine verranno presentati i due moduli principali del programma: quello client e quello server.

4.1 L'ambiente .NET di Microsoft e il linguaggio C#

Al fine di garantire piena compatibilità dell'applicativo con la maggior parte dei sistemi server aziendali, occorre scegliere un ambiente di sviluppo e, di conseguenza, un linguaggio di programmazione che sia largamente diffuso.

Quando si parla di server aziendali generalmente si fa riferimento a sistemi MS Windows. Secondo quanto riportato dall'ente di analisi IDC[1] nel primo trimestre del 2010, la quota di mercato relativa alla vendita di server di casa Microsoft è pari al 75,3%. Va considerato, inoltre, che l'indagine è stata effettuata su tutto il comparto server, includendo quindi gli apparati utilizzati in ambito Web, come Web server; in tale mercato, infatti, i sistemi Linux sono molto diffusi . Pertanto se l'analisi venisse ristretta ai soli server per il trattamento dei dati aziendali, (ad es. server utilizzati da sistemi ERP, CRM, ecc) la percentuale sarebbe di gran lunga maggiore.

Per queste ragioni UNOX ha scelto di sviluppare un applicativo compatibile con i sistemi operativi di casa Microsoft e di appoggiarsi all'ambiente .NET per l'implementazione. Questa scelta offre la possibilità di installare il servizio anche in SO

dedicati al mercato PC, (quali Windows XP, Windows Vista, Windows 7, ecc) garantendo quindi la compatibilità del sistema anche in instalazioni minori, cioè nei sistemi informativi di quei clienti che non sono dotati di apparati server ma soltanto di classici computer.

La suite di prodotti .NET è un progetto all'interno del quale Microsoft ha creato una piattaforma di sviluppo software, .NET, la quale è una versatile tecnologia di programmazione ad oggetti. La prima versione di .NET è stata rilasciata nel 2002. La sua caratteristica peculiare è di essere indipendente dalla versione operativa di Windows su cui è installata e di includere molte funzionalità progettate espressamente per integrarsi in ambiente Internet e garantire il massimo grado di sicurezza ed integrità dei dati.

.NET è corredato da una serie di strumenti di sviluppo delle applicazioni, progettati per funzionare in modo integrato all'interno della piattaforma. Uno dei principali strumenti è l'IDE, (Integrated Development Environment) denominato Visual Studio.

Con la nascita del framework .NET Microsoft ha introdotto sul mercato nuovi linguaggi di programmazione, tra cui:

- C#, linguaggio ad oggetti simile al Java della Oracle Corporation;
- Visual Basic .NET, linguaggio orientato agli oggetti e multi-threaded basato sulla sintassi di VisualBasic;
- J#, variante di J++ (la versione Microsoft di Java).

C#[2], fin dal suo esordio, è stato ben accolto dalla comunità di sviluppatori Microsoft, in quanto, in un certo senso, è il linguaggio che meglio degli altri descrive le linee guida sulle quali ogni programma .NET gira; questo linguaggio infatti è stato creato da Microsoft specificatamente per la programmazione in .NET. I suoi tipi di dati "primitivi" hanno una corrispondenza univoca con i tipi .NET e molte delle sue astrazioni, come classi, interfacce, delegati ed eccezioni, sono particolarmente adatte a gestire il .NET Framework.

Per queste ragioni, e per il fatto che UNOX dispone già di un bagaglio di esperienza nello sviluppo in tale linguaggio ha scelto proprio C# per lo sviluppo della sua nuova applicazione.

4.1.1 Applicazioni .NET in ambienti diversi da MS Windows

I linguaggi del Framework .NET sono linguaggi interpretati. Questo significa che a partire dal codice sorgente scritto ad esempio in C#, il compilatore integrato in Visual Studio produce un ulteriore listato, chiamato solitamente pseudo-codice. Il pseudo-codice prodotto viene poi letto dall'interprete .NET (CLR, Common Language Runtime), il quale lo traduce in linguaggio macchina in tempo reale durante l'esecuzione dei programmi.

Questa tecnica permette di rendere il software scritto completamente indipendente dall'architettura di sistema (SO e processore), in quanto il codice macchina viene prodotto runtime dall'interprete direttamente per l'architettura scelta.

Proprio per questo motivo negli anni sono nate delle piattaforme CLR indipendenti (cioè non sviluppate da Microsoft), in grado di eseguire codice .NET. In sostanza, attraverso un'opera di de-engineering si è riusciti a ricavare le regole del pseudo-linguaggio .NET e questo ha consentito di scrivere compilatori ed interpreti, ad esempio per sistemi Linux e MacOS.

Il CLR .NET indipendente universalmente più conosciuto e che negli anni ha riscosso molto successo è Mono, sviluppato da Novell. Mono è disponibile per moltissimi SO e architetture differenti, fra cui Windows, Linux e MacOS.

Pertanto, al fine di aumentare il grado di compatibilità del sistema, UNOX ha pensato di rendere il servizio di sistema "Mono-compatibile": durante le fasi di sviluppo del software sono stati effettuati dei test periodici per controllare che le librerie .NET utilizzate dal software siano presenti all'interno del CLR di Mono.

4.1.2 Struttura base di un servizio di sistema

Il questa sezione del testo verrà presentata e analizzata la struttura base di un servizio di sistema scritto in linguaggio C#[2], la quale è stata integrata nell'applicativo sviluppato al fine di rendere il programma compatibile con i servizi di sistema in ambiente MS Windows.

In ogni caso, come vedremo in seguito, modificando poche righe di codice è possibile rendere l'applicazione completamente equivalente ad un normale applicativo; di conseguenza sarà possibile configurarla per renderla compatibile con l'ambiente scelto.

Prima di procedere con l'analisi del codice C# diamo uno sguardo alle differenze che intercorrono fra un normale applicativo e un servizio di sistema:

- i servizi di sistema sono processi in grado di avviarsi e arrestarsi in modo completamente autonomo, senza bisogno di interventi da parte dell'utente; per questo motivo, come è già stato detto in precedenza, i servizi non sono provvisti di interfaccia utente;

- i servizi di sistema sono processi che vengono eseguiti dal SO anche se nel sistema la procedura di login non è (ancora) stata eseguita da nessun account utente;

- i servizi, a differenza degli applicativi standard, dispongono di una serie di funzioni, le quali permettono di avviare l'applicazione, di arrestarla, di metterla un uno stato di pausa, ecc.

Il .NET Framework mette a disposizione degli sviluppatori una serie di librerie che possono essere utilizzate all'interno dei programmi per implementare una varietà di funzioni differenti, come: strutture di gestione dei dati, funzioni di criptazione, strutture socket TCP, funzioni per la gestione di stringhe e numeri, ecc. Tutte le librerie sono scritte seguendo la logica di programmazione OOP (Object Oriented Programming) e sono contenute in una serie di file, detti *file assembly*. Pertanto, ogni qualvolta si intende far uso di una specifica libreria all'interno dei propri programmi,

bisogna includere all'interno del proprio progetto C# sia il riferimento al file assembly sia una direttiva *using* nell'header del codice sorgente del programma.

Il programmatore che intende scrivere un servizio di sistema deve far riferimento ad un'apposita classe, che dev'essere estesa dalla classe principale dell'applicativo scritto. La classe prende il nome di *ServiceBase* ed è contenuta nell'assembly *System.ServiceProcess* (che, come appena detto, va incluso nel progetto).

La struttura base di un servizio di sistema scritto in linguaggio C# segue l'impostazione del seguente codice:

```
1  using System;
2  using System.Diagnostics;
3  using System.ServiceProcess;
4
5  namespace WindowsService
6  {
7    class WindowsService : ServiceBase
8    {
9      /// <summary>
10     /// Costruttore
11     /// </summary>
12     public WindowsService()
13     {
14       this.ServiceName = "Nome del servizio";
15       this.EventLog.Log = "Nome del log del servizio";
16
17       // Flag per la gestione degli handle degli eventi
18       this.CanHandlePowerEvent = false;
19       this.CanHandleSessionChangeEvent = false;
20       this.CanPauseAndContinue = true;
21       this.CanShutdown = true;
22       this.CanStop = true;
23     }
24
25     /// <summary>
26     /// Main Thread
27     /// </summary>
28     static void Main()
29     {
30       ServiceBase.Run(new WindowsService());
31
32       /* Eventuale altro codice dedicato all'inizializzazione del
             programma */
33     }
34
35     /// <summary>
36     /// Dispose degli oggetti che lo richiedono
37     /// </summary>
38     protected override void Dispose(bool disposing)
39     {
40       base.Dispose(disposing);
41     }
42
43     /// <summary>
```

```
44      /// Codice di avvio dell'applicazione - avvio thread,
            inizializzazione variabili, ecc.
45      /// </summary>
46      protected override void OnStart(string[] args)
47      {
48        base.OnStart(args);
49      }
50
51      /// <summary>
52      /// Codice di arresto dell'applicazione - stop dei thread,
            salvataggio dati, ecc.
53      /// </summary>
54      protected override void OnStop()
55      {
56        base.OnStop();
57      }
58
59      /// <summary>
60      /// Codice dedicato all'arresto temporaneo dell'applicazione
61      /// </summary>
62      protected override void OnPause()
63      {
64        base.OnPause();
65      }
66
67      /// <summary>
68      /// Codice dedicato allo sblocco dell'applicazione in seguito ad un
            comando Pause
69      /// </summary>
70      protected override void OnContinue()
71      {
72        base.OnContinue();
73      }
74
75      /// <summary>
76      /// Codice da eseguire all'arresto del sistema operativo -
            salvataggio di dati sensibili, ecc.
77      /// </summary>
78      protected override void OnShutdown()
79      {
80        base.OnShutdown();
81      }
82
83      /// <summary>
84      /// Metodo dedicato all'invio di comandi al servizio senza l'
            utilizzo di socket o altri metodi di comunicazione
85      /// </summary>
86      /// <param name="command">Integer compreso fra 128 e 256</param>
87      protected override void OnCustomCommand(int command)
88      {
89        base.OnCustomCommand(command);
90
91        /* Codice di esecuzione del comando */
92      }
93    }
94  }
```

Nel costruttore (riga 12) è possibile definire il nome del servizio ed impostare una serie di flag che permettono di abilitare o meno gli handle per i rispettivi eventi.

Il servizio infatti può fare in modo di essere avvisato all'avvio, all'arresto, o ad esempio durante il cambio di stato dell'alimentazione. Se il flag relativo ad un determinato evento è abilitato, la funzione corrispondente verrà avviata e il servizio potrà svolgere determinate funzioni. Come è stato documentato nel codice di esempio, all'arresto del servizio è possibile arrestare i thread dell'applicazione, salvando prima eventuali dati sensibili. Come vedremo in seguito tali funzioni, sono state sfruttate anche dal servizio di URC.

Oltre ad estendere la classe *ServiceBase,* il servizio, per implementare correttamente le procedura di installazione nel sistema operativo, deve estendere anche un'altra classe, chiamata *Installer*.

Tale classe fa parte dell'assembly *System.Configuration.Install*.

Di seguito viene riportato il relativo codice di esempio:

```
1  using System;
2  using System.ComponentModel;
3  using System.Configuration.Install;
4  using System.ServiceProcess;
5
6  namespace WindowsService
7  {
8    [RunInstaller(true)]
9    public class WindowsServiceInstaller : Installer
10   {
11     /// <summary>
12     /// Costruttore per WindowsServiceInstaller
13     /// </summary>
14     public WindowsServiceInstaller()
15     {
16       ServiceProcessInstaller serviceProcessInstaller = new
             ServiceProcessInstaller();
17       ServiceInstaller serviceInstaller = new ServiceInstaller();
18
19       // Service Account Information
20       serviceProcessInstaller.Account = ServiceAccount.LocalSystem;
21       serviceProcessInstaller.Username = null;
22       serviceProcessInstaller.Password = null;
23
24       // Service Information
25       serviceInstaller.DisplayName = "Nome␣visualizzato␣dall'installer"
             ;
26       serviceInstaller.StartType = ServiceStartMode.Automatic;
27
28       // Il nome del servizio deve coincidere con quello definito nel
             costruttore di WindowsService
29       serviceInstaller.ServiceName = "Nome␣del␣servizio";
30
31       this.Installers.Add(serviceProcessInstaller);
32       this.Installers.Add(serviceInstaller);
33     }
34   }
```

```
35 }
```

Una volta compilato il programma con le due classi appena viste è possibile installare il servizio nel sistema operativo avviando il seguente batch file:

```
1  @ECHO OFF
2  REM Inserire la directory di installazione del Framework .NET
3  set DOTNETFX2=%SystemRoot%\Microsoft.NET\Framework\v2.0.50727
4  set PATH=%PATH%;%DOTNETFX2%
5
6  echo Installazione del servizio...
7  echo ---------------------------------------------------
8  InstallUtil /i NomeServizio.exe
9  echo ---------------------------------------------------
10 echo Fatto.
```

4.2 Struttura dell'applicazione

Dopo aver visionato un esempio di codice C# relativo alla scrittura e all'installazione di un servizio di sistema, passiamo ad analizzare i dettagli implementativi relativi al funzionamento dell'applicazione.

Come si è visto alla sezione 3.4, ogni nodo server è in grado di comunicare con i nodi di livello immediatamente inferiore e superiore della struttura ad albero. In particolare ogni nodo farà uso del proprio modulo client per contattare i propri nodi figli, mentre renderà disponibile il proprio modulo server per evadere le richieste provenienti dal proprio nodo padre. Si è poi stabilito che il passaggio dei dati fra moduli client e server di livelli diversi avvenga attraverso lo scambio di tabulati XML.

Ciò che non appare ancora chiaro, però, è come avvengono questi dialoghi, in particolare come ogni nodo possa elaborare nuove richieste da inviare ai nodi figli e come gestisca, invece, le richieste provenienti dal nodo padre.

Prima di tutto va detto che le richieste che vengono propagate nella struttura ad albero e che vengono gestite dai diversi nodi server possono essere di diversi tipi; questo, come è possibile immaginare, è conseguenza del fatto che il sistema deve permettere il recupero e la trasmissione verso i forni di dati differenti. Ad esempio, possiamo prevedere di avere una richiesta per la memorizzazione di un nuovo programma di cottura, una richiesta per il prelievo di un log HACCP e così via. Ciò che è importante capire è che a richieste diverse corrispondono XML diversi e che, come è naturale immaginare, che ad ogni richiesta ogni nodo server dovrà compiere azioni specifiche.

Una determinata richiesta, che da questo momento in poi chiameremo *attività*, può essere avviata ad un livello qualsiasi dell'albero su volontà dell'utente. In sostanza, accedendo all'interfaccia web-based di uno specifico nodo (che apparterrà quindi ad un determinato livello), l'utente può avviare una determinata procedura di prelievo dei dati, che a livello sistemistico corrisponde all'immissione nell'albero di un determinato XML. L'attività così generata verrà propagata, o forse è il caso di dire *instradata,* attraverso i diversi rami fino a raggiungere il forno di interesse.

Analizzando questo meccanismo in base a quanto detto finora, ciò che inizia a trasparire, però, è che una volta che un nodo avvia una determinata attività immettendola nell'albero, non potrà più conoscere l'esito dell'operazione. Ad esempio, supponiamo che un utilizzatore avvii una richiesta di scrittura di un nuovo programma di cottura attraverso l'interfaccia di un nodo a livello X; ciò che il nodo farà sarà avviare una nuova comunicazione con il nodo figlio connesso al sotto-ramo dell'albero collegato al forno di interesse per trasmettergli l'XML con il nuovo programma di cottura. Una volta che la comunicazione sarà terminata il nodo figlio a livello *X+1* eseguirà la stessa operazione, trasmettendo l'attività a livello *X+2*. A questo punto però il nodo che ha avviato l'attività non avrà più il controllo su ciò che è avvenuto e non avrà riscontri sull'effettivo risultato dell'operazione, in quanto non è a conoscenza di ciò che è accaduto dopo che la richiesta è stata instradata a livello inferiore (*X+1*). Questa mancanza rappresenta un grande limite in quanto l'utente che ha avviato l'attività di scrittura del nuovo programma di cottura non potrà sapere se il forno di ineteresse è stato effettivamente programmato, o se l'attività è stata abortita ad uno dei livelli sottostanti perchè, ad esempio, uno dei nodi è fuori servizio.

Quindi ciò che risulta chiaro è che occorre mettere in atto delle politiche di controllo delle diverse attività, per fare in modo che l'utente abbia la certezza che le operazioni compiute vengono portate a termine con successo.

Si può quindi pensare di fare in modo che, una volta che un nodo termina un'attività che gli è stata conferita da un nodo di livello superiore, comunichi l'esito dell'esecuzione al proprio padre, il quale può visualizzare il risultato sulle maschere di interfaccia utente.

Per mettere in atto questa semplice, ma efficace politica, l'unica difficoltà è rappresenrtata, come è stato visto alla sezione 3.2.2, dal fatto che le specifiche del sistema prevedono che ogni nodo conosca solo la lista dei propri figli; per tale motivo un nodo che vuole comunicare al padre l'esito dell'attività non può farlo autonomamente ad attività terminata, in quando non dispone di un metodo per avviare la comunicazione. Si potrebbe quindi pensare di modificare tale specifica ed integrare in ogni nodo altri due moduli: un client per la gestione delle comunicazioni con il padre ed un server per le richieste provenienti dai propri figli. Chiaramente l'ipotesi risulta alquanto dispendiosa, sia in termini di sviluppo (iniziale e per i futuri aggiornamenti), che di configurazione del sistema, in quanto il numero delle regole da applicare ai firewall di ogni livello verrebbe raddoppiato.

Durante le fasi di progettazione del sistema è stata trovata una soluzione alternativa, che prevede l'utilizzo di tabelle per la gestione delle attività.

4.2.1 Le tabelle di gestione delle attività

In linea del tutto generale possiamo dire che le attività che vengono gestite da un nodo possono essere di due tipi:

- esterne, quando le attività arrivano dal livello superiore su comunicazione del padre;

- interne, quando le attività hanno origine nel nodo stesso, ovvero quando è l'utente ad immetterle nel sistema attraverso l'applicativo di interfaccia proprio servendosi di quel nodo server.

Per ragioni analoghe un nodo può gestire una determinata attività applicando due politiche differenti:

- esecuzione, quando il nodo esegue l'attività perchè questa risulta indirizzata al nodo stesso (anche nel caso in cui l'attività sia stata immessa dall'utente direttamente nel nodo stesso);
- instradamento, quando l'attività risulta indirizzata ad uno dei figli e il nodo deve quindi propagarla a livello inferiore.

Un altro aspetto che va approfondito riguarda la comunicazione fra interfaccia utente e servizio di sistema che, come è stato detto più volte, avviene attraverso lo scambio di dati servendosi del database di appoggio.

In sostanza, una volta che l'applicativo di interfaccia ha ricevuto un comando da parte dell'utente deve comunicarlo al servizio di sistema, il quale dovrà avviare una nuova attività. La fase di passaggio delle consegne può essere quindi realizzata in due modi differenti:

- facendo in modo che l'interfaccia scriva nel database ciò che dev'essere fatto e che il servizio di sistema avvii la nuova attività generandone prima il relativo XML;
- facendo in modo che sia l'interfaccia stessa a generare l'XML relativo all'attività e che una volta generato lo inserisca in un'apposita tabella del database in modo che venga instradato dal servizio di sistema.

Chiaramente si può subito capire che conviene optare per la seconda soluzione in quanto la prima, a livello software, comporta un doppio passaggio di dati. Questo perché, partendo dal presupposto che è comunque necessaria una routine software dedicata alla generazione degli XML, nella seconda soluzione il comando dell'utente viene tradotto direttamente in XML e immesso nel sistema, mentre nella prima soluzione il comando viene prima tradotto in un linguaggio dedicato al dialogo fra interfaccia e servizio, per poi venire tradotto in XML dal servizio stesso. Dunque la prima soluzione, oltre che appesantire il lavoro di sviluppo, aumenta il carico di lavoro del servizio di sistema, senza portare alcun beneficio tangibile.

Scegliendo dunque la seconda soluzione, bisogna prevedere all'interno del database una tabella nella quale poter memorizzare le diverse attività, ognuna delle quali sarà corredata dal relativo XML. Di conseguenza il servizio di sistema dovrà essere dotato di una routine software che ad intervalli regolari analizzi tale tabella, per verificare la presenza di nuove attività da mandare in esecuzione. Naturalmente la tabella dovrà permettere di tenere traccia anche dello stato di esecuzione delle diverse attività, per consentire al servizio di sistema di riconoscere le attività già terminate. Lo stesso stato potrebbe quindi essere monitorato dall'applicativo di interfaccia, per comunicare all'utente utilizzatore l'effettiva esecuzione dei comandi.

La tabella in questione potrebbe essere utilizzata anche per inserire le attività che provengono dai nodi di livello superiore, le quali ai fini dell'esecuzione possono essere trattate allo stesso modo.

Ed è proprio qui che nasce l'idea che è stata adottata in URC per gestire efficientemente l'esecuzione delle attività e soprattutto per rilevare e memorizzare il risultato relativo all'esecuzione di ogni comando.

L'idea che sta alla base del meccanismo di verifica è molto semplice: si pensi di trattare le due tipologie di attività viste all'inizio di questa sezione del testo allo stesso modo e di assegnare ad ogni attività presente nella tabella del database una determinata chiave, la quale viene inserita anche nell'XML a corredo dell'attività. Così facendo quando un nodo a livello *X* provvederà ad instradare l'attività al livello inferiore *X+1* trasmettendo l'XML, con la richiesta trasmetterà implicitamente anche la chiave dell'attività, la quale potrà essere utilizzata anche a livello inferiore come chiave di identificazione. In questo modo, supponendo che il nodo a livello *X+1* sia a diretto contatto con il forno destinatario dell'attività, una volta che l'attività è stata eseguita dal bridge in modo corretto, il nodo (quello a livello *X+1*) potrà registrare l'esito positivo direttamente nella tabella delle attività del proprio database. A questo punto, ad intervalli regolari, il nodo di livello *X* può "chiedere" al nodo di livello *X+1* l'esito dell'operazione utilizzando la chiave condivisa e quest'ultimo potrà rispondere immediatamente andando a leggere lo stato dell'attività.

L'unico punto debole di questo meccanismo sta nella scelta delle chiavi da assegnare ad ogni attività, in quanto queste dovranno risultare univoche per tutti i livelli dell'albero. Le soluzioni a questa problematica possono essere molteplici e derivare da approcci differenti.

In URC il problema è stato risolto in questo modo: ogni chiave è costituita dalla concatenazione del nome attribuito al nodo (vedi specifica n. 10 alla sezione 3.2.2) e di un numero. Ereditando le proprietà della caratteristica legata al nome del nodo, per garantire l'univocità della chiave a tutti i livelli basterà fare in modo che la chiave, o meglio il numero scelto, sia univoco all'interno della tabella del nodo che genera l'attività.

Queste tecniche consentono quindi di far fronte alle due problematiche analizzate, da un lato realizzando una verifica sull'effettiva esecuzione delle attività e dall'altro gestendo in modo semplificato le richieste provenienti dall'applicativo di interfaccia.

4.2.2 Esecuzione delle sotto-attività

Dopo aver trovato una soluzione ai problemi affrontati nella sezione precedente, passiamo ad analizzare il modo in cui il servizio di sistema gestisce l'esecuzione delle attività presenti nel proprio database.

Prima di fare ciò va approfondito un aspetto legato agli XML di trasmissione dei dati, anche se finora non è ancora stato visto un esempio di come questi vengano effettivamente rappresentati. Questa anticipazione risulta però necessaria per approfondire le tematiche trattate in questa sezione.

In base a quanto è stato detto in precedenza ogni file XML contiene dati di un determinato tipo; i file XML trattati del sistema saranno dunque molteplici ed in particolare ne avremmo uno per ogni differente tipologia di richiesta.

Analizziamo ora un caso pratico: supponiamo di avere un sistema URC che controlla 100 forni e di avviare la scrittura di un nuovo programma di cottura alla posizione 5 di ogni forno. In base a quanto detto finora l'applicativo di interfaccia dovrà generare un XML contenente il nuovo programma, la posizione di memorizzazione e l'elenco dei forni da aggiornare. Tale XML, una volta inserito nell'apposita tabella del database, verrà instradato ai livelli inferiori dell'albero fino ad arrivare alle foglie (forni). Considerando che il comando riguarda tutti i nodi del sistema, è naturale che durante il suo instradamento venga propagato su tutti i sotto-rami dell'albero.

In questo caso come vengono gestiti gli stati di conferma di avvenuta esecuzione del comando? In base a quanto detto in precedenza la tabella contenente le attività da eseguire riporta un'apposita colonna per la memorizzazione dello stato. Ma se l'attività in questione coinvolge più di un nodo?

Si è appena visto che nel caso in cui un nodo sia collegato a più figli l'attività madre deve essere suddivisa in sotto-attività, al fine di propagare la richiesta a tutti i sotto-rami dell'albero interessati. Pertanto è possibile prevedere una seconda tabella nel database di sistema (relazionata alla prima) che permetta di associare ad ogni attività madre l'elenco delle attività legate ad ogni nodo figlio. Naturalmente nel caso in cui l'attività madre coinvolga un solo nodo figlio tale tabella conterrà una sola riga di riferimento. Ogni riga della tabella, oltre all'XML della singola sotto-attività, può quindi contenere anche lo stato di esecuzione del comando, il quale, in caso di problemi, permette di identificare l'effettiva causa.

4.2.3 Timeout, ritrasmissioni e politiche di priorità

Una volta che un nodo inserisce nel sistema una determinata attività non si ha alcuna garanzia del fatto che questa venga eseguita. I motivi per cui questo potrebbe succedere sono molteplici e di seguito se ne riportano alcuni a titolo d'esempio:

- problemi di rete nelle comunicazione fra nodo e nodo (ad esempio troppi pacchetti TCP persi);
- nodi momentaneamente fuori servizio per malfunzionamenti dell'hardware, o per blocchi al software;
- saturazione di un nodo a causa di troppe attività da gestire.

In linea generale un'attività può essere abortita per due differenti tipologie di guasti:

- temporanei, quando il disservizio dura una frazione di secondo o al massimo qualche decina di secondi (es. problemi di comunicazione);
- permanenti, quando il guasto richiede l'intervento di un tecnico per poter essere risolto e la durata del disservizio supera qualche decina di secondi (es. guasto hardware).

Per come è strutturato URC i guasti (di qualsiasi tipo) che non permettono di portare a compimento le varie attività rappresentano un problema serio, in quanto un disservizio ad un nodo collocato ai livelli superiori dell'albero può compromettere l'utilizzo di una grossa parte del sistema. Per questo risulta importante implementare delle politiche di gestione delle attività che rendano il sistema tollerante ai guasti. Chiaramente se ciò non può essere fatto per i disservizi permanenti, nel caso di interruzioni temporanee si può fare in modo che le attività non vengano abortite prematuramente.

Per questo motivo URC adotta due tipologie di timeout sulle attività, unite a delle politiche di gestione delle ritrasmissioni. Il sistema prevede, inoltre, dei meccanismi di gestione delle priorità, per evitare che un nodo saturi le proprie capacità operative in seguito all'esecuzione di troppe attività onerose in termini di risorse.

Timeout globale dell'attività All'aggiunta di una determinata attività nel database di appoggio viene inserito anche un apposito valore di timeout. Tale valore, che a livello implementativo è un numero intero, rappresenta il numero di secondi entro il quale l'attività deve essere portata a termine; se il timeout dovesse scadere l'attività verrebbe automaticamente abortita.

Questo accorgimento permette di evitare che in nodo rimanga in attesa di ricevere una risposta dai propri figli bloccando l'esecuzione dell'attività a tutti i livelli dell'albero.

Chiaramente, essendo il servizio di sistema identico in tutti i nodi server e considerando che le attività vengono gestite in modo indipendente dai diversi nodi, tale meccanismo viene applicato a tutti i livelli dell'albero.

Timeout sulle singole sotto-attività (retry-limit) Come per il timeout globale dell'attività madre, anche ogni sotto-attività è legata ad uno specifico timeout il cui valore, però, ha un significato differente.

Mentre nel caso precedente il valore intero rappresenta una soglia temporale entro cui l'attività dev'essere compiuta, in questo caso il timeout identifica il numero di tentativi che un nodo ha a disposizione per cercare di eseguire il comando. Questo tipo di timeout è stato denominato *retry-limit*. Se la procedura di esecuzione dell'attività (ovvero il dialogo con un nodo figlio o con uno dei bridge) non può essere avviata, o dovesse terminare prematuramnte a causa di un disservizio, il nodo ripeterà da capo la procedura di esecuzione. Ad ogni tentativo il servizio di sistema provvederà a decrementare il valore di timeout retry-limit, fino a che non si raggiungerà lo zero. In questo caso l'attività verrà considerata non compiuta e verrà abortita definitivamente.

Va evidenziato che non vi è alcuna relazione fra i due tipi di timeout e che la loro scadenza reciproca è arbitraria e dipende dal caso specifico che si viene a creare.

Quello che è certo è che il timeout che scade per primo è quello che determina la fine dell'attività.

Priorità di esecuzione delle sotto-attività Alcune delle attività viste alla sezione 2.3 risultano più "impegnative" di altre, sia in termini di risorse, sia per quanto concerne il tempo dedicato alla loro esecuzione.

Ad esempio, la procedura di aggiornamento del firmware è probabile che impegni i nodi server a diretto contatto con i forni per molto tempo, soprattutto nel caso in cui un singolo nodo sia collegato ad un numero elevato di bridge. Considerando ad esempio un firmware di 200-300 kByte e tenendo in considerazione le limitate capacità computazionali dell'hardware interno al bridge, è probabile che il nodo server impieghi qualche decina di secondi per effettuare il trasferimento dell'intera attività. Se il nodo server che esegue l'attività dovesse effettuare la stessa operazione in 20 forni diversi, l'attività di aggiornamento nel complesso potrebbe impiegare anche più di 10 minuti.

Per tale motivo, supponendo che un nodo server possa gestire un numero limitato di attività simultanee, è necessario implementare delle politiche di gestione delle priorità per l'esecuzione delle diverse attività. Se ciò non viene fatto, nel caso dell'esempio precedente, si corre il rischio che nuove attività che vengono assegnate al nodo, in seguito all'avvio dell'aggiornamento del firmware vengano abortite per il sopraggiungere del timeout globale.

URC per la gestione delle attività prevede due livelli di priorità: alta priorità e bassa priorità. L'attribuzione dei due livelli di priorità viene riportata in Tabella 4.1.

Attività	Priorità
Aggiornamento firmware	Bassa
Programmi di cottura	Alta
Parametri di funzionamento	Alta
Log HACCP	Bassa
Log programmi di cottura avviati	Bassa
Warning e allarmi	Alta
Risultati dei programmi di autodiagnosi	Bassa

Tabella 4.1: elenco delle attività con il relativo grado di priorità

In pratica, supponendo che ogni nodo server possa eseguire 10 attività simultanee[1], la logica interna al servizio di sistema prevede che possano essere gestite simultaneamente al massimo 5 attività a bassa priorità (attività più onerose). In sostanza il sistema riserva (almeno) 5 "posizioni" per l'esecuzione di attività ad alta priorità, per fare in modo che anche in presenza di molte attività onerose queste non possano compromettere l'operatività del sistema.

Inoltre il sistema mette in atto una seconda politica di priorità, basata sulla profondità di origine delle attività. In sostanza ogni qualvolta un nodo deve avviare l'esecuzione di una determinata attività dà priorità ai comandi che provengono dai livelli più alti. Questo meccanismo fa sì che richieste più "profonde" vengano gestite più velocemente man mano che il numero di nodi interessati aumenta, consentendo di liberare più rapidamente le risorse occupate di ogni livello.

[1] come vedremo in seguito questo sarà proprio il valore di default utilizzato dal programma

4.3 Il database MySQL

Dopo aver analizzato nel dettaglio tutti gli elementi implementativi relativi al servizio di sistema, continuiamo la trattazione rivolgendo l'attenzione alla struttura del database di appoggio.

Come è stato detto più volte i dati gestiti dai diversi nodi server vengono memorizzati all'interno di una base di dati, la quale assolve anche la funzione di canale di comunicazione fra servizio di sistema e applicativo di interfaccia utente in base ai meccanismi discussi alla sezione 4.2.1.

L'applicativo software che permette di gestire in modo efficiente la mole di dati di URC e, più in generale, di tutti i sistemi informatici che necessitano di archiviare e reperire informazioni, è il Database Management System.

Il DBMS è un sistema software progettato per consentire la creazione e manipolazione efficiente di database (ovvero di collezioni di dati strutturati) solitamente da parte di più utenti. I DBMS svolgono un ruolo fondamentale in numerose applicazioni informatiche: dalla contabilità, alla gestione delle risorse umane e alla finanza, fino a contesti tecnici, come la gestione di rete, o la telefonia. Se in passato i DBMS erano diffusi principalmente presso le grandi aziende e istituzioni (le quali potevano permettersi l'impegno economico derivante dall'acquisto delle grandi infrastrutture hardware necessarie per realizzare un sistema di database efficiente), oggi il loro utilizzo è diffuso praticamente in ogni contesto.

Un DBMS può essere costituito da un insieme assai complesso di programmi software che controllano l'organizzazione, la memorizzazione e il reperimento dei dati (campi, record e archivi) in un database. Un DBMS controlla anche la sicurezza e l'integrità del database e accetta richieste di dati da parte del programma applicativo per "istruire" successivamente il sistema operativo per il trasferimento dei dati appropriati.

Nel caso di URC il DBMS utilizzato dovrà essere di tipo relazionale (RDBMS), in quanto i dati contenuti nelle diverse tabelle dovranno essere collegati fra loro secondo il costrutto entità-relazione.

Oggigiorno sul mercato informatico si trovano moltissime implementazioni di DBMS, sia gratuite che a pagamento.

Fra i database gratuiti MySQL è utilizzato nella maggior parte delle applicazioni software di alto livello.

MySQL è un RDBMS composto da un client con interfaccia a caratteri e un server, entrambi disponibili sia per sistemi Unix come GNU/Linux sia per MS Windows. Dal 1996 supporta la maggior parte della sintassi SQL e si prevede che in futuro adotti il pieno rispetto dello standard ANSI. Possiede delle interfacce per diversi linguaggi, compreso un driver ODBC, due driver Java, un driver per Mono e .NET.

I motivi che hanno spinto UNOX a scegliere MySQL come supporto alla gestione dei dati del sistema sono riassunti di seguito:

- prezzo: MySQL viene rilasciato gratuitamente per applicazioni gratuite; per le applicazioni a pagamento è previsto invece l'acquisto di una licenza, che in ogni caso presenta un prezzo davvero contenuto;

- prestazioni: MySQL presenta ottime prestazioni e per il suo funzionamento non sono richiesti particolari requisiti hardware;
- dialetto SQL: come suggerisce il nome MySQL adotta la sintassi SQL, che ormai rappresenta a tutti gli effetti uno standard in ambito informatico. L'utilizzo del dialetto SQL permette, in caso di necessità, di migrare ad un'altra base di dati in poco tempo e senza dover effettuare particolari revisioni al codice sorgente del programma.

4.3.1 Progettazione della base di dati

Iniziamo ora l'analisi realtiva alla progettazione del database per la parte che interessa il servizio di sistema; in seguito la trattazione verrà completata anche per la parte che riguarda la sola interfaccia utente. Nel database troveremo infatti delle tabelle che verranno utilizzate solo dall'applicativo di interfaccia.

Come concordato con l'azienda (vedi sezione 1.3), durante l'attività di tesi si è implementata la sola parte del sistema dedicata alla gestione dei programmi di cottura. Per tale motivo la struttura delle tabelle necessarie al salvataggio di altri tipi di dati, come ad esempio log HACCP, o parametri di sistema non è stata definita. In ogni caso è possibile prevedere che tale attività non comporti un particolare impegno, in quanto l'implementazione delle altre attività richiede l'adozione di singole tabelle specifiche (ad es. una tabella per i log HACCP, una tabella per salvare i parametri di funzionamento e così via).

La struttura del database dovrà permettere l'implementazione di tutte le routine software come sono state presentate nelle precedenti sezioni del testo, in modo che il risultato sia conforme alle specifiche del sistema.

La progettazione di una base di dati, in linea generale, prevede le seguenti fasi:

1. definizione dei requisiti informativi;
2. definizione dello schema concettuale;
3. ristrutturazione delle schema concettuale;
4. definizione dello schema logico relazionale.

La definizione dei requisiti informativi è già stata svolta approfonditamente nelle sezioni precedenti e lo studio ci ha permesso di delineare le entità principali del sistema, ovvero:

- nodi dell'albero
- attività
- sotto-attività

Al fine di riuscire ad implementare la gestione dei *programmi di cottura*, all'elenco precedente va aggiunta anche questa entità.

Un altro aspetto che può essere ricavato dalla trattazione sin qui svolta sta nell'esigenza di disporre di una funzione per il salvataggio delle preferenze di programma

proprie di ogni nodo server. Considerando che tali opzioni in alcuni casi dovranno essere recuperate sia dal servizio di sistema sia dall'applicativo di interfaccia, si può pensare di sfruttare proprio il database MySQL come supporto per il salvataggio.

A partire da queste considerazioni si può passare alla definizione dello schema concettuale. Il risultato è visibile in Figura 4.1.

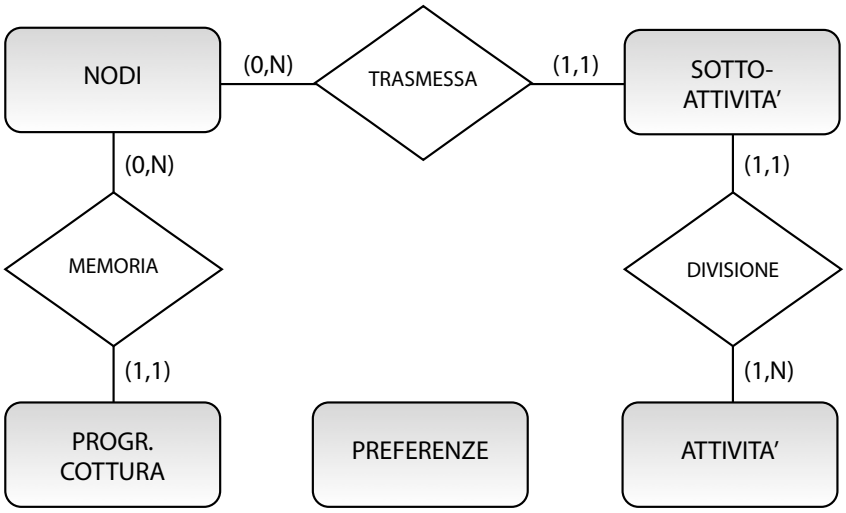

Figura 4.1: Schema concettuale del database (parte relativa al servizio di sistema).

Lo schema mette in risalto le entità principali fra loro collegate attraverso un'opportuna relazione; ogni relazione viene regolamentata da un'opportuna cardinalità minima e massima. I legami evidenziati dallo schema sono i seguenti:

- attività - sotto-attività: ogni attività madre viene opportunamente *divisa* in sotto-attività ed in particolare in almeno una sotto-attività, che nel caso di singoli comandi può coincidere con l'intera attività madre;
- nodi - sotto-attività: ogni sotto-attività viene *trasmessa* ad uno specifico nodo e nello specifico ad uno e uno soltanto; ogni nodo invece può essere ineteressato da zero, o più sotto-attività;
- nodi - programmi di cottura: ogni nodo *memorizza* zero, o più programmi di cottura; ogni programma di cottura residente in tabella viene invece memorizzato all'interno di un singolo nodo.

Lo schema concettuale va ora popolato inserendo gli attributi di ogni entità e di ogni relazione. Considerando però che nel caso specifico non compaiono attributi composti né altre particolarità, quali, ad esempio, le gerarchie e che il mapping da schema concettuale a schema logico relazione è immediato, passiamo all'analisi direttamente della struttura tabellare interna al database.

La Figura 4.2 riporta l'insieme delle tabelle MySQL che compongono il database di URC. Ad un'analisi sommaria possiamo subito individuare gli attributi che permettono di implementare le funzionalità descritte alle sezioni precedenti del testo. Procediamo quindi con l'analisi dettagliata di ogni tabella.

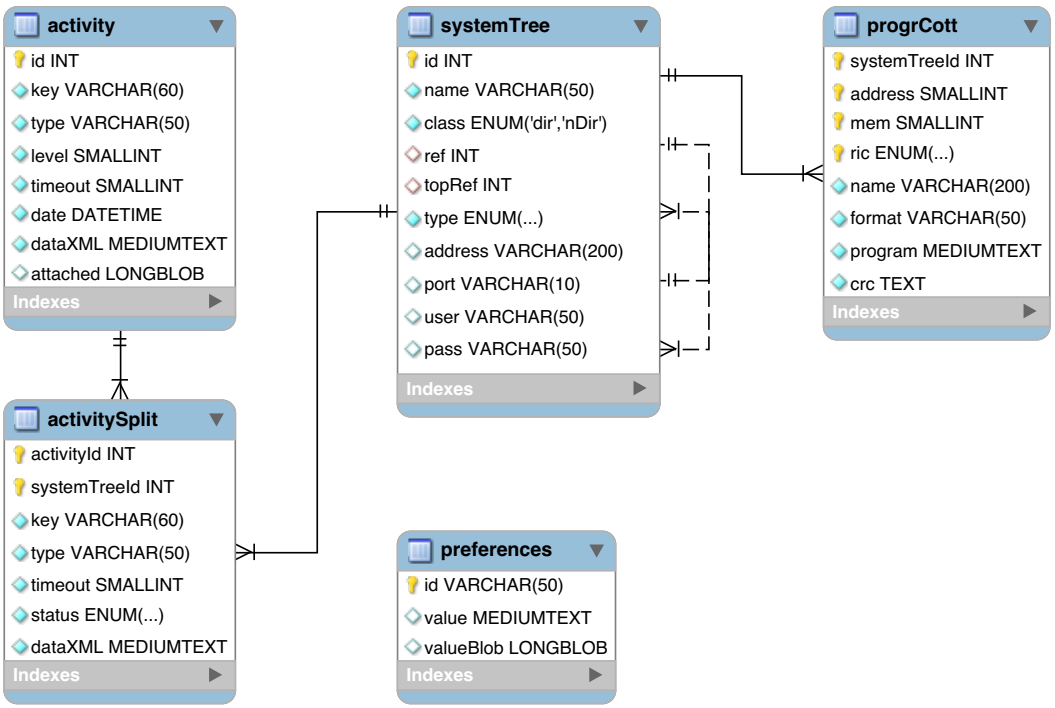

Figura 4.2: Tabelle del database MySQL di appoggio di supporto al servizio di sistema.

4.3.1.1 Preferences

Questa tabella consente di archiviare e recuperare le preferenze relative ad ogni singolo nodo, tra le quali troviamo il nome attribuito al nodo stesso in fase di configurazione. Il nome, che deve essere univoco all'interno del sistema, (per garantire la generazione di chiavi di attività uniche) viene impostato attraverso l'applicativo di interfaccia utente (come vedremo in seguito, attraverso un'oppurtuna maschera di gestione delle preferenze). La tabella però è già stata predisposta per poter contenere informazioni di altro tipo, come ad esempio dati binari. Per questo, oltre alla colonna *id*, la quale costituisce la chiave di ogni tupla, ovvero il nome attribuito alla preferenza, troviamo altre due colonne: *value* e *valueBlob*. I due attributi consentono di memorizzare, rispettivamente, preferenze testuali (stringhe) e dati binari (immagini, file, ecc). Di seguito si riporta il codice SQL di creazione della tabella:

```
1 CREATE TABLE IF NOT EXISTS `preferences`
2 (
3   `id` VARCHAR(50) BINARY NOT NULL ,
4   `value` MEDIUMTEXT NULL ,
5   `valueBlob` LONGBLOB NULL ,
6   PRIMARY KEY (`id`)
7 ) ENGINE = InnoDB;
```

Come si evince dal codice SQL i campi *value* e *valueBlob* possono anche essere *NULL*; questo perché i campi saranno alternativamente vuoti in base al tipo di preferenza memorizzata (una preferenza testuale avrà il campo *valueBlob* uguale a *NULL* e viceversa).

4.3.1.2 SystemTree

La tabella *systemTree* permette ad ogni nodo di salvare l'elenco di tutti i suoi discendenti. In sostanza la tabella permette di archiviare sia le informazioni relative ai figli diretti, (direttamente raggiungibili o di livello immediatamente inferiore) sia quelle relative a tutti i nodi dell'albero raggiungibili gerarchicamente attraverso i figli diretti.

Come è già stato detto, durante la fase di configurazione iniziale di ogni nodo, l'amministratore di sistema (cioè colui che si occupa di installare e configurare URC presso il cliente utilizzatore) dovrà inserire in questa tabella l'elenco dei figli diretti, cioé l'elenco delle sedi aziendali di livello gerarchico inferiore. Chiaramente l'inserimento delle entry in tabella, come per tutti gli altri dati, verrà fatto attraverso l'applicativo di interfaccia utente. Come vedremo in seguito, dopo che l'amministratore ha inserito tali informazioni potrà ottenere l'elenco di tutti i nodi di livello inferiore in modo automatico eseguendo un'opportuna attività. Tale operazione, inoltre, potrà essere svolta periodicamente e permetterà agli utenti utilizzatori di tenere aggiornata la struttura ad albero del sistema in ogni nodo, senza dover intervenire manualmente sui dati; ciò tornerebbe utile nel caso in cui si verificassero delle modifiche fisiche all'impianto (ad esempio in seguito all'aggiunta di un forno, o di un nuovo punto vendita).

Il codice SQL di creazione della tabella è il seguente:

```
1  CREATE TABLE IF NOT EXISTS `systemTree`
2  (
3    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
4    `name` VARCHAR(50) BINARY NOT NULL ,
5    `class` ENUM('dir','nDir') NOT NULL ,
6    `ref` INT UNSIGNED NULL ,
7    `topRef` INT UNSIGNED NULL ,
8    `type` ENUM('device','server') NOT NULL ,
9    `address` VARCHAR(200) BINARY NULL ,
10   `port` VARCHAR(10) BINARY NULL ,
11   `user` VARCHAR(50) BINARY NULL ,
12   `pass` VARCHAR(50) BINARY NULL ,
13   PRIMARY KEY (`id`) ,
14   UNIQUE INDEX `name_UNIQUE` (`name` ASC) ,
15   INDEX `refSystemTree` (`ref` ASC) ,
16   INDEX `topRefSystemTree` (`topRef` ASC) ,
17   CONSTRAINT `refSystemTree`
18     FOREIGN KEY (`ref`)
19     REFERENCES `systemTree` (`id`)
20     ON DELETE CASCADE
21     ON UPDATE NO ACTION,
22   CONSTRAINT `topRefSystemTree`
23     FOREIGN KEY (`topRef`)
```

```
24      REFERENCES `systemTree` (`id`)
25      ON DELETE CASCADE
26      ON UPDATE NO ACTION
27  ) ENGINE = InnoDB;
```

Come è stato detto più volte il nome attribuito ad ogni nodo deve essere univoco, pertanto può essere naturale pensare di utilizzarlo come chiave primaria della tabella. Il codice SQL evidenzia però che la chiave è costituita dal campo *id*, il quale è di tipo intero (unsigned) auto-increment. Questa scelta deriva dal fatto che le chiavi primarie di tipo intero sono molto più performanti di chiavi testuali e considerando che la tabella in questione dovrà essere relazionata alla tabella di archiviazione delle sotto-attività, (vedi schema concettuale a pagina 46) questa scelta consente di abbattere i tempi di esecuzione delle query. In ogni caso il codice SQL evidenzia una direttiva *UNIQUE* per il campo *name*; pertanto il vincolo relativo all'unicità del nome di un nodo viene correttamente preservato.

Continuando l'analisi del codice di creazione della tabella troviamo i seguenti campi:

- *class*: indica la classe di appartenenza del nodo (diretta o indiretta, rispettivamente *dir* e *nDir*), ovvero se il nodo in questione è in figlio diretto o indiretto;
- *type*: indica il tipo di nodo (nodo *server* o forno - colonna di forni, *device*);
- *address* e *port*: indicano rispettivamente indirizzo IP (o nome DNS) e porta TCP del nodo; tali campi saranno *NULL* nel caso di figli indiretti, in quanto le comunicazioni verranno veicolate attraverso i nodi (figli) di livello direttamente inferiore dell'albero;
- *user* e *pass*: credenziali di autenticazione del nodo server richieste per l'avvio delle comunicazioni nei dialoghi di trasmissione delle nuove attività.

Le ultime due colonne della tabella, che non sono ancora state analizzate, sono *ref* e *topRef*. Come si può notare analizzando il codice SQL, questi due attributi sono relazionati alla chiave primaria della tabella stessa.

L'attributo *ref* consente di memorizzare il padre di un determinato nodo, consentendo di ricostruire la struttura ad albero del sistema. L'attributo viene usato, ad esempio, nell'applicativo di interfaccia utente per visualizzare l'albero dei nodi, in modo che l'utente abbia un riscontro tangibile sull'organizzazione gerarchica del sistema. Naturalmente le tuple relative ai figli diretti avranno *ref* pari a *NULL*.

L'attributo *topRef* consente di memorizzare invece il super-padre che nella struttura ad albero è figlio diretto del nodo. Ad esempio, supponiamo di avere un sistema con nodo radice Europa, collegato ai nodi Italia e Germania. Supponiamo inoltre che nel sistema sia presente anche il nodo Lombardia, collegato ad Italia al livello superiore e al nodo foglia Malpensa al livello inferiore. Se analizziamo la tupla del nodo Malpensa nel database relativo al nodo radice Europa, troveremo che la colonna *topRef* punta al nodo Italia. La funzione dell'attributo è quindi facilmente

comprensibile: se dal nodo Europa vogliamo contattare il nodo Malpensa, dobbiamo inviare la richiesta al nodo Italia, il quale a sua volta invierà la richiesta al nodo presente in *topRef* della tupla Malpensa del proprio database, ovvero Lombardia. Nel caso in cui si debbano inviare comandi ai nodi dei livelli inferiori dell'albero in strutture altamente verticalizzate, l'attributo *topRef* consente di ottenere subito il nodo a cui instradare le attività, senza dover quindi ricostruire di volta in volta la struttura dell'albero attraverso l'attributo *ref*. In sostanza l'attributo *topRef* non è indispensabile, ma consente di risparmiare risorse computazionali ogni qualvolta si deve instradare una determinata attività ai livelli inferiori.

Un'ultima nota riguarda un importante dettaglio, evidenziato nello script SQL di creazione della tabella alle linee 20 e 25. In sostanza le direttive SQL in questione permettono di eliminare i nodi collegati ad un determinato nodo nel momento in cui quest'ultimo viene eliminato dalla struttura ad albero. Riprendendo l'esempio precedente, eliminando il nodo Lombardia verrebbe eliminato automaticamente dal DBMS anche il nodo Malpensa, oltre a tutti gli altri nodi collegati in modo diretto e indiretto a Lombardia. Questo processo permette di eseguire query di tipo *DELETE* che nelle tabelle *InnoDB* di MySQL non potrebbero essere eseguite nel caso di tuple con relazioni pendenti.

4.3.1.3 Activity

La tabella *activity*, come suggerisce il nome, permette di memorizzare l'elenco delle attività da eseguire. La tabella rappresenta il canale di comunicazione primario fra applicativo di interfaccia utente e servizio di sistema: il primo vi inserisce le tuple delle attività da eseguire e il secondo provvede ad instradarle ai relativi nodi del sistema e a raccoglierne i risultati.

Il codice SQL di creazione della tabella è il seguente:

```
1  CREATE TABLE IF NOT EXISTS `activity`
2  (
3    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
4    `key` VARCHAR(60) BINARY NOT NULL ,
5    `type` VARCHAR(50) BINARY NOT NULL ,
6    `level` SMALLINT UNSIGNED NOT NULL ,
7    `timeout` SMALLINT UNSIGNED NOT NULL ,
8    `date` DATETIME NOT NULL ,
9    `dataXML` MEDIUMTEXT NOT NULL ,
10   `attached` LONGBLOB NULL ,
11   PRIMARY KEY (`id`) ,
12   UNIQUE INDEX `key_UNIQUE` (`key` ASC)
13 ) ENGINE = InnoDB;
```

Come nel caso dell'attributo *name* della tabella *systemTree*, l'attributo *key* della tabella *activity*, che rappresenta la chiave univoca relativa all'attività, non è chiave primaria. Anche in questo caso, infatti, la chiave dell'attività è testuale (concatenazione del nome del nodo e di un numero casuale) e le performance di un indice relativo a valori di questo tipo, paragonato ad un indice di interi, sono nettamente

inferiori. Pertanto, come si è fatto per la tabella *systemTree*, la chiave della tabella *activity* è un intero, *id*, e l'attributo *key* viene imposto di tipo *UNIQUE*.

Gli altri attributi hanno la seguente funzione:

- *type*: memorizza la stringa relativa al tipo di richiesta (ad esempio, come vedremo in seguito, *setProgrCott* e *getProgrCott* per indicare rispettivamente la scrittura e la lettura dei programmi di cottura); l'attributo permette al servizio di sistema di identificare velocemente il tipo di attività al fine di applicare, ad esempio, le politiche di priorità analizzate in Tabella 4.1;
- *level*: identifica la profondità di instradamento dell'attività espressa in numero di livelli dell'albero; l'attributo è necessario per applicare le politiche di priorità analizzate alla sezione 4.2.3. Man mano che l'attività passa di livello in livello, l'attributo *level* viene incrementato di una unità;
- *timeout*: valore di timeout di esecuzione dell'attività espresso in secondi;
- *date*: data e ora di inserimento dell'attività; come vedremo in seguito, utile per ripulire la tabella delle vecchie attività già eseguite;
- *dataXML*: campo testuale contenente l'XML con la richiesta che dev'essere eseguita; si ricorda che l'XML in questione contiene l'intera attività così come è stata memorizzata dall'applicativo di interfaccia utente, o così come è stata ricevuta dai livelli superiori;
- *attached*: campo binario che permette di salvare dati a corredo dell'attività (essenzialmente è dedicato all'archiviazione delle nuove versioni del software nelle attività di aggiornamento firmware).

4.3.1.4 ActivitySplit

Questa tabella permette di salvare l'elenco delle sotto-attività collegate ad una particolare attività madre. L'SQL di creazione della tabella è il seguente:

```
1  CREATE TABLE IF NOT EXISTS `activitySplit`
2  (
3    `activityId` INT UNSIGNED NOT NULL ,
4    `systemTreeId` INT UNSIGNED NOT NULL ,
5    `key` VARCHAR(60) BINARY NOT NULL ,
6    `type` VARCHAR(50) NOT NULL ,
7    `timeout` SMALLINT UNSIGNED NOT NULL ,
8    `status` ENUM('busy','complete','error') NOT NULL ,
9    `dataXML` MEDIUMTEXT NOT NULL ,
10   PRIMARY KEY (`activityId`, `systemTreeId`) ,
11   INDEX `activitySplitActivityId` (`activityId` ASC) ,
12   INDEX `activitySplitSystemTreeId` (`systemTreeId` ASC) ,
13   UNIQUE INDEX `key_UNIQUE` (`key` ASC) ,
14   CONSTRAINT `activitySplitActivityId`
15     FOREIGN KEY (`activityId` )
16     REFERENCES `activity` (`id` )
17     ON DELETE CASCADE
```

```
18      ON UPDATE NO ACTION,
19    CONSTRAINT `activitySplitSystemTreeId`
20      FOREIGN KEY (`systemTreeId`)
21      REFERENCES `systemTree` (`id`)
22      ON DELETE CASCADE
23      ON UPDATE NO ACTION
24  ) ENGINE = InnoDB;
```

Gli attributi *activityId* e *systemTreeId*, che insieme costituiscono la chiave primaria della tabella, sono chiavi esterne collegate rispettivamente alla tabella *activity* e *systemTree*. In sostanza ogni sotto-attività viene collegata alla propria attività madre e al nodo verso cui deve essere instradata.

Gli attributi *key* e *type* hanno invece lo stesso significato dei medesimi attributi presenti nella tabella *activity*.

Il campo *timeout* contiene invece il timeout della sotto-attività espresso in numero di tentativi di esecuzione rimanenti, così come riportato alla sezione 4.2.3.

dataXML contiene invece l'XML relativo alla sotto-attività, che in pratica non sarà altro che un estratto dell'XML dell'attività madre contenente le sole informazioni che riguardano la sotto-attività in questione.

L'ultimo attributo da analizzare, *status*, permette di salvare lo stato di esecuzione della sotto-attività. L'attributo è di tipo *ENUM* e può valere:

- *busy*: quando l'attività non è ancora stata avviata, o non è ancora stata ricevuta una risposta dall'host remoto;
- *complete*: quando l'attività è stata eseguita e portata a termine con successo;
- *error*: quando l'attività è stata avviata, ma è stata abortita prematuramente a causa di un errore, come ad esempio lo scadere del timeout (numero massimo di tentativi).

Come per la tabella *systemTree,* anche in questo caso le due chiavi esterne presentano la direttiva *ON DELETE CASCADE* per fare in modo che in caso di eliminazione di un nodo, o di un'attività madre, il DBMS elimini anche tutte le sotto-attività a loro collegate.

Un particolare che va evidenziato sta nell'assenza dell'attributo *attached*, che è presente solo nella tabella relativa all'attività madre. Questa scelta è giustificata dal fatto che, nel caso di attività di aggiornamento firmware, la nuova versione del software è la stessa per tutti i dispositivi inclusi nella richiesta; pertanto il nuovo firmware può essere recuperato direttamente nella riga dell'attività madre.

In ultima analisi va evidenziato un altro dettaglio: la tabella *activity* non presenta l'attributo *status*, che compare invece in ogni sotto-attività. Questa scelta implica che lo stato globale di esecuzione di un'attività madre può essere ricavato a partire dagli stati delle singole sotto-attività. In pratica un'attività madre risulta:

- terminata con successo se tutti gli stati delle singole sotto-attività sono pari a *complete*

- ancora in fase di eleborazione se almeno uno degli stati delle sotto-attività è pari a *busy*
- terminata in modo errato se vi è almeno uno stato delle sotto-attività pari ad *error*.

4.3.1.5 ProgrCott

ProgrCott è la tabella dedicata al salvataggio dei programmi di cottura dei forni. In sostanza l'insieme delle tuple presenti in tabella (una tupla per ogni programma di cottura) non sono altro che l'esatta copia della memoria interna di tutti i forni collegati al sistema. Di seguito si riporta il codice SQL di creazione della tabella:

```
1  CREATE TABLE IF NOT EXISTS `progrCott`
2  (
3    `systemTreeId` INT UNSIGNED NOT NULL ,
4    `address` SMALLINT UNSIGNED NOT NULL ,
5    `mem` SMALLINT UNSIGNED NOT NULL ,
6    `ric` ENUM('utente','preimpostata') NOT NULL ,
7    `name` VARCHAR(200) BINARY NOT NULL ,
8    `format` VARCHAR(50) NOT NULL ,
9    `program` MEDIUMTEXT NOT NULL ,
10   `crc` TEXT NOT NULL ,
11   PRIMARY KEY (`systemTreeId`, `address`, `mem`, `ric`) ,
12   INDEX `progrCottSystemTreeId` (`systemTreeId` ASC) ,
13   CONSTRAINT `progrCottSystemTreeId`
14     FOREIGN KEY (`systemTreeId` )
15     REFERENCES `systemTree` (`id` )
16     ON DELETE CASCADE
17     ON UPDATE NO ACTION
18 ) ENGINE = InnoDB;
```

La chiave primaria è formata dai seguenti attributi:

- *systemTreeId*: nodo del sistema che memorizza il programma di cottura;
- *address*: indirizzo sul bus Modbus del dispositivo che memorizza il programma di cottura. Come è stato detto più volte in una colonna possono essere presenti più forni (e altri dispositivi), pertanto, considerando che la relazione con la tabella *systemTree* consente di identificare solo la colonna, il campo *address* identifica il dispositivo che memorizza questo specifico programma;
- *mem*: posizione di memorizzazione - numero di programma che compare nel display della maschera comandi del forno;
- *ric*: tipo di programma di cottura, può essere *utente* o *preimpostata*.

Gli altri attributi della tabella hanno le seguenti funzioni:

- *name*: nome del programma di cottura, lo stesso che compare nella maschera comandi del forno;

- *format*: formato di scrittura dei dati del programma di cottura. Attualmente il campo non è necessario in quanto UNOX utilizza un solo formato di rappresentazione dei dati, ma in futuro potrebbe servire per identificare, ad esempio, la versione;

- *program*: contiene il tabulato con il programma di cottura vero e proprio. Considerando che all'interno del forno il programma di cottura è un array di byte prima di essere trasmesso via XML, e poi salvato in questa tabella, dovrà essere convertito in stringa (ad esempio 1 byte può essere tradotto con l'equivalente espressione composta da una coppia di caratteri esadecimali);

- *crc*: contiene il codice CRC di controllo del contenuto di *program*.

4.3.2 Connessione a MySQL da C#

Per fare in modo che il servizio di sistema (scritto in linguaggio C#) possa prelevare e scrivere i dati nel database MySQL di appoggio, occorre mettere a punto una serie di routine software, che permettano di stabilire una connessione con il DBMS e che supportino l'esecuzione di query in linguaggio SQL.

Per effettuare l'accesso a MySQL da una qualche routine software, il DBMS mette a disposizione diversi canali di comunicazione, tra cui ODBC ed una serie di driver specifici per i linguaggi può usati.

Per l'ambiente .NET di Microsoft il consorzio MySQL mette a disposizione dei programmatori un'apposita libreria, chiamata *MySQL .NET Connector*, ovvero un connettore scritto appositamente per questo ambiente e che integra una serie di funzioni per l'interfacciamento totale alla base di dati. La particolarità che risulta senza dubbio più interessante è la piena integrazione del connettore con le più comuni classi del .NET, prima fra tutte la classe *DataTable*.

In C# i dati strtturati in forma tabellare possono essere manipolati attraverso una serie di classi standard dell'ambiente, le quali prendono il nome di DataTable e DataSet. Le due classi sono state sviluppate pensando all'organizzazione dei dati interna ad un database, le tabelle create possono contenere una serie di colonne identificate da un nome e che possono contenere una specifica tipologia di dato (che può essere un dato primitivo dell'ambiente .NET o una qualsiasi classe), proprio come avviene nelle base di dati.

Pertanto l'equivalente di una tabella di un database in C# viene rappresentata dalla classe DataTable, la quale sarà costituita da un insieme di colonne di tipo *DataColumn* (ovvero l'insieme degli attributi dela tabella); le varie tuple della tabella in C# corrispondo ad un insieme di oggetti di tipo *DataRow*. L'intero database, ovvero l'insieme delle diverse tabelle, in C# viene rappresentato dalla classe *DataSet*, la quale non è altro che un contenitore di oggetti DataTable.

Va evidenziato che la convenienza nell'utilizzare queste specifiche classi sta nel fatto che l'accesso e la modifica dei dati così strutturati via codice C# avviene in modo molto intuitivo. Inoltre la classe DataTable mette a disposizione una serie di metodi che introducono funzioni avanzate, quali ad esempio l'ordinamento e la conversione dei dati in formato XML.

Per comprendere la potenza e la flessibilità delle classi appena presentate possiamo analizzare il seguente esempio.

Supponiamo di avere stabilito una connessione con il database MySQL e di aver prelevato tutte le entry della tabella *systemTree*. Come abbiamo detto precedentemente, in C# la tabella viene mappata in un oggetto *DataTable*. A questo punto se volessimo accedere al contenuto della colonna *name* della prima riga di tipo *device* estratta, il codice C# sarebbe il seguente:

```
1  DataTable systemTree;
2
3  /* Codice per la connessione al database MySQL e l'esecuzione della
       query "SELECT name FROM systemTree WHERE type='device'" per il
       caricamento dei dati nel DataTable systemTree */
4
5  string nomeNodo;
6  nomeNodo = systemTree.Rows[0]["name"];
```

Se volessimo invece generare un XML con l'elenco dei nomi dei nodi del sistema il codice sarebbe il seguente:

```
1  DataTable systemTree;
2  string xml;
3
4  /* Codice per la connessione al database MySQL e l'esecuzione della
       query "SELECT name FROM systemTree" per il caricamento dei dati nel
       DataTable systemTree */
5
6  xml = "<elencoNodi>";
7  foreach (DataRow node in systemTree.Rows)
8  {
9    xml += "<nodo>";
10   xml += node["name"];
11   xml += "</nodo>";
12 }
13 xml += "</elencoNodi>";
```

Un risultato equivalente si sarebbe potuto ottenere utilizzando i metodi standard della classe DataSet per la conversione delle strutture dati in formato XML. Un esempio di codice è il seguente:

```
1  DataTable systemTree;
2  systemTree.TableName = "nodo";
3  string xml;
4
5  /* Codice per la connessione al database MySQL e l'esecuzione della
       query "SELECT name FROM systemTree" per il caricamento dei dati nel
       DataTable systemTree */
6
7  // Creazione DataSet
8  DataSet elenco;
```

```
9  elenco.DataSetName = "elencoNodi";
10 // Inserimento del DataTable systemTree nel DataSet
11 elenco.Tables.Add(systemTree);
12
13 // Mappatura del DataSet in XML
14 StringWriter sw = new StringWriter();
15 elenco.WriteXml(sw);
16 xml = sw.ToString();
17 sw.Close();
```

che dà come risultato il seguente XML (il risultato è contenuto nella variabile *xml*):

```
1  <elencoNodi>
2    <nodo>
3      <name>Italia</name>
4    </nodo>
5    <nodo>
6      <name>Germania</name>
7    </nodo>
8    ...
9    <nodo>
10     <name>Malpensa</name>
11   </nodo>
12 </elencoNodi>
```

Si può quindi capire come l'utilizzo delle strutture DataSet e DataTable permetta di risparmiare molto tempo nella scrittura del codice e permetta di generare listati facilmente comprensibili perché autoesplicativi.

Classe URCDatabase Il connettore .NET per MySQL mette a disposizione del programmatore una serie di primitive per la gestione delle connessioni e per l'esecuzione delle query.

Se all'interno del servizio di sistema si dovesse scrivere una routine C# per il prelievo di qualche dato dal database il codice generato conterrebbe diverse chiamate alle primitive del connettore per la generazione e la manipolazione di altrettanti oggetti. Tale codice dovrebbe essere ripetuto ogni qualvolta si rendesse necessario il prelievo o la modifica di qualche dato, pertanto, anche in previsione di eventuali interventi di revisione, risulta senz'altro opportuno racchiuderlo in dei metodi standard che possono essere richiamati in caso di necessità.

Per questo in URC si è realizzata una classe, chiamata URCDatabase, la quale contiene una serie di metodi di alto livello, che mascherano la routine di interfacciamento al database; tali metodi al loro interno fanno uso delle primitive messe a disposizione da MySQL .NET Connector.

Si riporta di seguito l'interfaccia della classe:

```
1  public static class URCDatabase
2  {
```

```
3     // Parametri di connessione al database
4     public struct ConnectionParams
5     {
6       public string host, port, schema, username, password;
7     }
8     public static ConnectionParams connectionParams;
9     // Codici di ritorno del metodo SaveResult
10    public enum ResultCode { Execute, ZeroModify, ConcurrencyError,
          GenericError };
11
12    // Memorizza i dati connessione nella variabile oggetto
          connectionParams, cerca di stabilire una connessione con il
          database e ritorna il risultato dell'operazione
13    public static bool TestConnection(string host, string port, string
          user, string password, string schema);
14    // Esegue una query di tipo SELECT, passata attraverso l'oggetto di
          tipo MySqlCommand, e ritorna un oggetto di tipo URCDatabaseResult
15    public static URCDatabaseResult GetResult(MySqlCommand command);
16    // Salva le modifiche apportate all'oggetto di tipo URCDatabaseResult
           ottenuto da una precedente invocazione del metodo GetResult e ne
          ritorna il risultato
17    public static ResultCode SaveResult(URCDatabaseResult result);
18    // Esegue una query di tipo INSERT e ne ritorna il risultato
19    public static bool Insert(MySqlCommand command);
20    // Esegue una query di tipo DELETE e ne ritorna il risultato
21    public static bool Delete(MySqlCommand command);
22    // Esegue una query di tipo UPDATE e ne ritorna il risultato
23    public static bool Update(MySqlCommand command);
24    // Ritorna un oggetto DateTime del .NET contenente la data e l'ora
          del server MySQL
25    public static DateTime GetCurrentTime();
26  }
27
28  public class URCDatabaseResult
29  {
30    // Costruttore
31    public URCDatabaseResult();
32    // Get/Set dell'oggetto DataTable di C# con il risultato della query
          di SELECT
33    public DataTable dataTable;
34  }
```

La funzione di ogni metodo è riportata nei commenti all'interno del codice.

La classe utilizza un metodo di connessione al database di tipo *non persistente*; questo significa che prima di eseguire ogni nuova query le funzioni (come *GetResult* o *Insert*) stabiliscono una nuova connessione, che poi viene automaticamente chiusa prima che le funzioni vengano terminate. Questa tecnica evita l'utilizzo di connessioni *persistent*i, le quali vengono instaurate all'avvio del programma e vengono terminate al suo arresto, che potrebbero venire interrotte inaspettatamente durante il funzionamento del servizio di sistema e causarne l'arresto incontrollato (solitamente per il sopraggiungere di un'eccezione). Queso meccanismo comporta però l'avvio di un nuovo processo di connessione ad ogni nuova query e potrebbe portare ad uno spreco di risorse con conseguente rallentamento di esecuzione del codice.

Fortunatamente MySQL permette di risolvere l'inconveniente tramite l'utilizzo di un apposita funzionalità, detta di *pooling*, la quale prevede che il DBMS mantenga sempre delle connesioni attive (anche in caso di disconnessione esplicita dell'host remoto) per fare in modo che le pratiche di connesione possano essere svolte più rapidamente. Per attivare il pooling è sufficiente inserire un'opportuna direttiva nella stringa di connessione inviata alle primitive del connettore .NET per MySQL; questa procedura viene svolta automaticamente da tutte le funzioni della classe URCDatabase.

Un altro aspetto da approfondire riguarda l'utilizzo dell'oggetto *MySqlCommand*, che fa parte delle classi messe a disposizione del connettore di MySQL.

Per chiarire l'utilizzo di questa classe e della classe URCDatabase appena presentata si riporta un esempio di codice che svolge per intero uno degli esempi analizzati in precedenza e che permette di leggere il nome del primo nodo di tipo *device* estratto dalla tabella systemTree del database:

```
1  DataTable systemTree;
2  string tipoNodo = "device";
3
4  // Connessione al database
5  bool connStatus = URCDatabase.TestConnection("localhost", "42110", "
       root", "root", "urc");
6  if (!connStatus)
7    return;
8  // Estrazione dei nodi di tipo device
9  MySqlCommand command = new MySqlCommand("SELECT␣name␣FROM␣systemTree␣
       WHERE␣type=?type");
10 command.Parameters.AddWithValue("?type", tipoNodo);
11 URCDatabaseResult result = URCDatabase.GetResult(command);
12 systemTree = result.dataTable;
13
14 // Lettura del nome del nodo
15 string nomeNodo = systemTree.Rows[0]["name"];
```

In sostanza la classe MySqlCommand rappresenta l'interfaccia che permette di gestire tutte le query SQL in modo tale che i parametri passati da codice C# vengano mappati nell'equivalente tipo di dato di MySQL. La classe utilizza un sistema di associazione *chiave-valore*, dove ogni chiave presente all'interno della query (la quale comincia sempre con un punto interrogativo) viene convertita in valore dalla funzione *AddWithValue*. La classe permette quindi di gestire la composizione delle query in modo più veloce e risolve in modo automatico tutti i problemi derivanti dall'utilizzo di caratteri particolari per MySQL, quali ad esempio l'apostrofo, o l'apice.

4.4 Specifiche XML

L'XML è un metalinguaggio che permette di definire nuovi linguaggi, i quali a loro volta permettono di organizzare i dati in una specifica struttura. L'XML offre la possibilità di definire infiniti nuovi linguaggi e, di conseguenza, infinite nuove strutture. Ogni linguaggio viene progettato per rappresentare dati di uno specifico tipo

e le regole del linguaggio sono definite proprio sulla base dei dati che devono essere strutturati.

Un listato XML è composto da una serie di *tag*. Un tag rappresenta un particolare marcatore che inizia con il simbolo di *minore* "<" e termina con il simbolo di *maggiore* ">"; all'interno dei due simboli compare una stringa di testo, che inizia immediatamente dopo il simbolo di apertura tag, la quale rappresenta il *nome* del tag. Ogni tag può contenere zero o più *attributi* che permettono di descrivere il tag stesso. Un esempio di tag con attributi è il seguente: *<persona sesso="M" cittadinanza="Ita">*.

In un file XML ogni tag che viene aperto deve essere anche chiuso, e il marcatore di chiusura riporta lo stesso nome del tag con anteposto il simbolo "/". Ad esempio il tag *<nome>* dovrà precedere il relativo tag di chiusura *</nome>*.

Un'altra regola fondamentale nella costituzione di un XML è l'incapsulamento dei tag. In sostanza fra i marcatori di apertura e chiusura di uno specifico tag possono essere inseriti altri tag, come nell'esempio seguente:

```
1  <elenco>
2    <persona sesso="M">
3      <nome>Alberto</nome>
4      <cognome>Bianco</cognome>
5    </persona>
6    <persona sesso="F">
7      <nome>Giulia</nome>
8      <cognome>Brown</cognome>
9    </persona>
10 </elenco>
```

Nell'esempio si nota come il tag *elenco* contenga al proprio interno il tag *persona*, il quale a sua volta contiene i tag *nome* e *cognome*.

Come è stato detto, XML permette di definire nuovi linguaggi ognuno dei quali presenterà regole differenti. Quindi XML non impone che i dati vengano strutturati in un modo piuttosto che in un altro: nell'esempio precedente per indicare il sesso della persona si sarebbe potuto usare un apposito tag al posto di inserire l'informazione nell'attributo *sesso* del tag *persona*.

Inoltre l'esempio mette in luce che ogni tag può essere ripetuto più volte all'interno dello stesso listato.

In URC, come è facile immaginare, ogni diversa attività fa uso di uno specifico XML che incapsula i dati dell'attività nel modo più appropriato.

Di seguito procediamo con l'analisi delle specifiche XML relative ad ogni attività implementata durante l'attività di tirocinio, descrivendo nel dettaglio la funzione di ogni singolo tag. Le specifiche relative a tutte le altre attività sono comunque simili; le differenze sostanziali, come vedremo in seguito, riguarderanno principalmente i listati XML di risposta, i quali avranno una struttura tale da poter descrivere le informazioni trasferite.

4.4.1 Rigenerazione dell'albero

Questa richiesta consente ad un qualsiasi nodo di recuperare la lista dei nodi presenti a livelli sottostanti, rigenerando così la struttura ad albero memorizzata nella tabella *systemTree* del proprio database e dei database di tutto il sotto-albero.

Questo tipo di attività viene eseguita ricorsivamente, propagando cioè la richiesta di livello in livello fino a raggiungere i nodi server di livello superiore rispetto alle foglie.

L'XML utilizzato per eseguire la richiesta è il seguente:

```
1 <getChildren>
2   <request>
3     <key>Italy-1548988665</key>
4     <level>2</level>
5   </request>
6 </getChildren>
```

L'XML è strutturato su 3 livelli. Questa caratteristica, che come vedremo in seguito è comune a tutte le richieste, deriva dal fatto che in C# questo tipo di rappresentazione può essere mappata su una struttura di tipo DataSet (vedi sezione 4.3.2). Nel caso in esame il DataSet avrà nome *getChildren* e conterrà una sola tabella chiamata *request*; la tabella avrà due colonne, *key* e *level,* e conterrà una sola riga con la coppia di dati (*Italy-1548988665*, *2*). Strutturando gli XML in questo modo, sarà possibile accedere alle direttive della richiesta senza bisogno di implementare parser specifici per ogni tipo di attività.

La richiesta *getChildren* contiene due direttive: *key* e *level.* La prima corrisponde alla chiave che è stata attribuita all'attività (definita dal nodo di livello superiore, o dall'applicativo di interfaccia utente), mentre la seconda corrisponde al livello di pofondità di propagazione della richiesta.

In risposta a questo tipo di richiesta possiamo avere due XML differenti. Il primo viene inviato dal destinatario della richiesta per indicare che l'attività non è ancora stata eseguita, o non è ancora terminta:

```
1 <busy />
```

Il secondo XML viene invece inviato nel caso in cui l'esecuzione dell'attività sia terminata e il destinatario della richiesta debba trasmettere i dati al mittente:

```
1  <children>
2    <request>
3      <key>Italy-1548988665</key>
4    </request>
5    <response>
6      <nodeName>NordEst</nodeName>
7      <status>complete | error</status>
8    </response>
9    <child>
10     <name>Limena1</name>
```

```
11      <ref>NordEst</ref>
12      <type>device</type>
13    </child>
14    <child>
15      <name>Padova2</name>
16      <ref>NordEst</ref>
17      <type>device</type>
18    </child>
19  </children>
```

Come è possibile vedere, il primo XML contiene solo il tag *busy* che, mappato in C#, corrisponde ad un DataSet vuoto con lo stesso nome. In questo caso l'inserimento di altri tag annidati non avrebbe senso, in quanto il nodo destinatario deve solo comunicare che "è occupato"; inoltre questa scelta consente di risparmiare banda in fase di trasmissione (che nel caso dell'invio dati da parte del bridge risulta comunque preziosa).

Il secondo XML risulta invece più corposo. Il significato dei tag che lo compongono visto nella relativa mappatura C# è il seguente:

- tabella *request*: usata per ritrasmettere la chiave relativa alla richiesta
- tabella *response*: contiene il nome del nodo che invia la risposta e lo stato di esecuzione dell'attività (completata con successo, o terminata in modo errato)
- tabella *child*: contiene i dati dei nodi collegati al destinatario
  - campo *name*: nome del nodo
  - campo *ref*: padre del nodo - equivalente al campo *ref* della tabella *systemTre* del database
  - campo *type*: tipo di nodo, forno (o colonna di forni), o server

Un dettaglio importante è la ritrasmissione del nome del nodo destinatario (tabella *response*). Sebbene questo sia conosciuto di default dal nodo mittente perché viene inserito dall'amministratore di sistema direttamente dall'interfaccia utente, è necessario che venga ritrasmesso in quanto potrebbe venire modificato in qualsiasi momento. Per mettere in atto, quindi, una completa rigenerazione dell'albero ad ogni livello, occorre che anche questa informazione venga ritrasmessa ad ogni nuova richiesta di tipo *getChildren*.

4.4.2 Scrittura dei programmi di cottura

Come si può immaginare, la struttura dell'XML relativo alla scrittura dei programmi di cottura conterrà nel listato di richiesta l'elenco dei programmi completo dell'indicazione relativa ai nodi da programmare, mentre nel listato di risposta verranno passate solo le informazioni relative all'esito dell'operazione.

L'XML di richiesta assume quindi la seguente struttura:

```
<setProgrCott>
  <request>
    <key>Italy-1548988665</key>
    <level>2</level>
  </request>
  <progrCott>
   [...]
  </progrCott>
</setProgrCott>
```

La tabella *request* riporta gli stessi dati della richiesta *getChildren*. La differenza sostanziale è la presenza della tabella *progrCott* dedicata al trasferimento vero e proprio dei programmi di cottura.

La struttura di tale tabella verrà analizzata nel seguito del testo poiché, come vedremo, verrà utilizzata anche negli XML relativi all'attività di lettura dei programmi di cottura.

Come per la richiesta *getChildren,* anche la richesta *setProgrCott* prevede due XML di risposta. Oltre al singolo tag *busy*, che assume lo stesso significato già visto in precedenza, troviamo il seguente XML:

```
<setProgrCottT>
  <request>
    <key>Italy-1548988665</key>
  </request>
  <response>
    <status>complete | error</status>
  </response>
</setProgrCottT>
```

Anche in questo caso il significato delle tabelle *request* e *response* è lo stesso dell'attività di rigenerazione dell'albero, con la sola differenza sul nome del DataSet.

Infine va evidenziato che la richiesta *setProgrCott* viene propagata nella struttura ad albero fino a raggiungere le foglie (ovvero i dispositivi da programmare).

4.4.3 Lettura dei programmi di cottura

La procedura di lettura dei programmi di cottura è stata implementata in modo tale che il nodo mittente della richiesta indichi quali programmi desidera prelevare. Ad esempio, il mittente può decidere di prelevare i programmi di cottura alla posizione 25 di tutti i forni, oppure di prelevare i programmi di cottura con nome "Pollo". In sostanza il mittente dovrà poter effettuare una specie di *query* nella quale verranno indicati i diversi parametri di ricerca.

Per uniformare i diversi listati XML e per semplificare le procedure di costruzione ed interpretazione delle attività si è deciso di fare in modo che l'XML utilizzato per il trasferimento dei programmi di cottura possa venire utilizzato anche per effettuare le interrogazioni di prelievo.

Per tale ragione l'XML relativo alla richiesta di lettura dei programmi di cottura assume la seguente struttura:

```
<getProgrCott>
  <request>
    <key>Italy -1548988665</key>
    <level>2</level>
  </request>
  <progrCott>
   [...]
  </progrCott>
</getProgrCott>
```

Mentre l'XML di risposta (oltre al solito *busy*) sarà formato nel modo seguente:

```
<getProgrCottT>
  <request>
    <key>Italy -1548988665</key>
  </request>
  <response>
    <status>complete | error</status>
  </response>
  <progrCott>
   [...]
  </progrCott>
</getProgrCottT>
```

Come nel caso della richiesta *setProgrCott,* anche questa attività viene propagata di livello in livello fino a raggiungere le foglie dell'albero.

4.4.4 XML di trasferimento dei programmi di cottura

Questo listato XML, come anticipato in precedenza, viene utilizzato nelle richieste *setProgrCott* e *getProgrCott* (e relativa *getProgrCottT*) per due differenti scopi: trasferimento dei dati e query.

La struttura dell'XML è la seguente:

```
<progrCott>
  <nSer />       // numero di serie del prodotto UNOX (univoco)
  <nodeName />   // nome del nodo destinatario
  <address />    // indirizzo del dispositivo sul bus
  <mem />        // posizione di memorizzazione
  <ric />        // tipo di ricetta (utente, preimpostata, ecc)
  <format />     // formato dati programma
  <sch />        // tipo di scheda dest. (forno, abbattit., ecc)
  <name />       // nome del programma di cottura
  <program />    // programma di cottura
  <crc />        // crc di controllo del programma
<progrCott />
```

Il significato di ogni campo è indicato nei commenti presenti alla destra di ogni tag. La maggior parte dei tag ha lo stesso significato degli omonimi attributi della colonna *progrCott* del database di sistema, a differenza dei seguenti:

- *nSer*: permette di contraddistinguere il dispositivo desiderato utilizzando il suo numero di serie; per le ragioni analizzate alla sezione 2.1.3 tale funzionalità non può essere al momento implementata, ma è stata prevista per le successive versioni del sistema;
- *nodeName*: indica il nome del nodo da programmare, o dal quale si desidera prelevare il/i programma/i di cottura;
- *sch*: questo campo viene utilizzato solo in fase di scrittura e permette di specificare il tipo di scheda elettronica destinataria (o tipo di dispositivo).

Come è possibile immaginare, mentre in caso di programmazione (richiesta *setProgrCott*) tutti i campi sono obbligatori, nel caso di lettura dei programmi di cottura ogni XML conterrà solo i tag necessari alla scrittura della query. Ad esempio, se dal nodo radice si vogliono prelevare i programmi di cottura di *tutti* i *forni* memorizzati alla posizione *25* l'XML sarà il seguente:

```
1 <progrCott>
2   <mem>25</mem>
3   <sch>forno</sch>
4 <progrCott />
```

L'esempio mette in evidenza che nel caso di interrogazione globale il tag *nodeName* può essere omesso. In tal caso ogni nodo server che riceve l'attività provvederà a generare una serie di sotto-attività, una per ogni nodo figlio.

L'esempio seguente illustra invece l'XML di prelievo di *tutti* i programmi di cottura della colonna di forni con nome *Padova3*:

```
1 <progrCott>
2   <nodeName>Padova3</nodeName>
3 <progrCott />
```

4.5 Modulo server

Dopo aver presentato l'ambiente di sviluppo, le politiche di gestione delle attività e il modo in cui queste vengono archiviate e trasmesse, passiamo all'analisi del codice di implementazione di uno dei due principali blocchi del programma: il modulo server.

Come è già stato detto più volte, il servizio di sistema dispone di due moduli, quello client e quello server, dedicati al dialogo rispettivamente dei nodi dell'albero di livello inferiore e superiore. I dialoghi avvengono secondo il noto paradigma client-server, nel quale il server rimane in attesa di connessioni in entrata da parte del client, il quale provvede ad avviarle in caso di necessità.

Nel caso di URC le comunicazioni avvengono attraverso lo scambio di listati XML, listati che per le richieste vengono memorizzati all'interno della base di dati MySQL. I listati di riposta, invece, vengono generati in tempo reale dal server, il quale li adatta di volta in volta alle direttive contenute nella richiesta che ha raccolto.

In URC il modulo server è costituito da un *thread* autonomo che viene avviato all'interno della funzione di avvio del servizio. La scelta di incapsulare il server in un thread separato si è resa necessaria per riuscire ad implementare delle routine di gestione delle connessioni *non bloccanti*, in modo tale da poter gestire più richieste di connessione simultaneamente e per fare in modo che il codice di gestione del server non blocchi l'esecuzione del modulo client. Alcune funzioni di gestione dei socket TCP, ad esempio quelle di accettazione delle connessioni in ingresso, risultano infatti bloccanti: ciò significa che finché l'interprete C# non ha terminato di eseguire il codice di tali funzioni, l'esecuzione delle altre parti del programma risulta bloccata. Un modo per ovviare a questo inconveniente sta nell'utilizzare thread separati per gestire l'esecuzione delle diverse parti del programma, che di conseguenza risulteranno completamente indipendenti.

In URC la classe che implementa il modulo server è stata chiamata *UNOXRemoteControlServer* e il codice C# di avvio del modulo server è il seguente:

```
1  if (urcServer != null && urcServer.IsAlive)
2    urcServer.Abort();
3  UNOXRemoteControlServer server = new UNOXRemoteControlServer();
4  urcServer = new Thread(new ThreadStart(server.Start));
5  urcServer.Start();
```

Il codice deve essere inserito all'interno del metodo *OnStart* dedicato all'avvio del servizio di sistema. Il listato riporta la classica procedura di avvio dei thread in C#. Dapprima viene verificato se il thread è già in esecuzione attraverso l'uso della proprietà *IsAlive* e se il controllo dà esito positivo si procede con l'arresto del thred. Successivamente al costruttore della classe *ThreadStart* viene passato il riferimento al metodo di avvio della classe UNOXRemoteControlServer (metodo *Start* dell'oggetto *server*) e per finire il nuovo thread viene avviato attraverso il metodo *Start*.

4.5.1 Schema a blocchi

La struttura di funzionamento del modulo server del servizio di sistema viene raffigurata nello schema a blocchi di Figura 4.3.

Il modulo si avvia con l'esecuzione del metodo *Start*, il quale è dedicato alla creazione di un oggetto *TcpListener*, come evidenziato nel seguente codice C#:

```
1  TcpListener myListener = new TcpListener(IPAddress.Any, options.Default
       .serverPort);
2  myListener.Start();
3  myListener.BeginAcceptSocket(new AsyncCallback(ConnectionCallback),
       myListener);
```

L'oggetto viene inizializzato per accettare connessioni provenienti da qualsiasi indirizzo IP ed indirizzate ad una specifica porta TCP; il valore della porta viene recuperato dalle opzioni di programma, le quali vengono impostate dall'amministratore di sistema. Naturalmente il relativo nodo padre di livello superiore dovrà essere configurato per inviare le richieste a questa specifica porta.

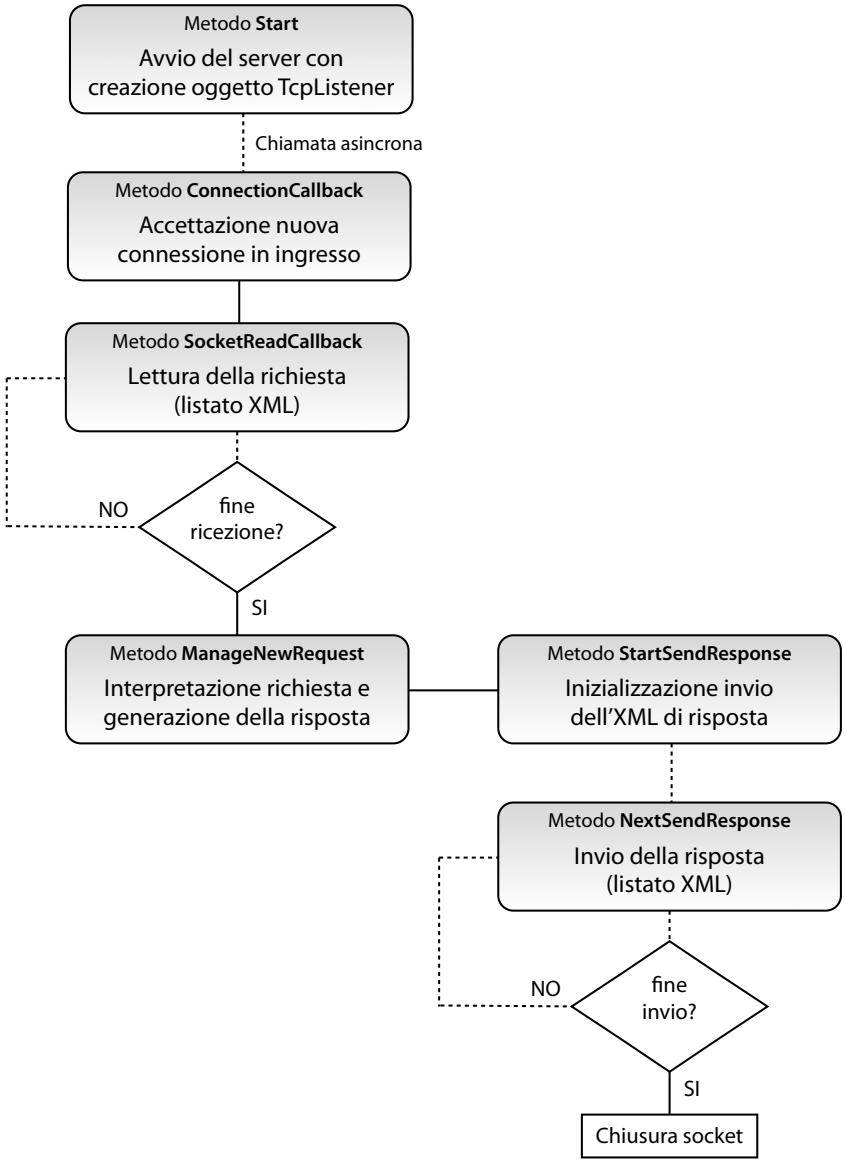

Figura 4.3: Schema a blocchi del modulo server del servizio di sistema.

L'oggetto *myListener* attende le connessioni in entrata in modo asincrono (non bloccante - metodo *BeginAcceptSocket*) e all'arrivo di una nuova richiesta provvede ad invocare il metodo *ConnectionCallback* sul socket TCP che è stato creato.

Tale metodo, quando viene invocato, può recuperare l'oggetto *myListener* dai parametri di ingresso ed iniziare la procedura di lettura asincrona dallo stream dati collegato al socket. Dopo aver avviato la lettura l'oggetto, *myListener* viene messo di nuovo in ascolto per accettare nuove connessioni in ingresso (in questo modo il modulo server può gestire più connessioni simultaneamente).

La fase di lettura salva i dati ricevuti in un oggetto di tipo *StringBuilder* e continua fino a quando la funzione viene invocata, anche se il flusso di stream è

vuoto (condizione di fine ricezione).

A questo punto viene invocata la funzione *ManageNewRequest*, che converte la stringa ricevuta in un oggetto di tipo *DataSet* e ne interpreta i parametri. In base al tipo di attività richiesta la funzione provvede a generare il relativo XML di risposta ed inizia l'invio dei dati al socket con l'invocazione della funzione *StartSendResponse*. La struttura e il codice della funzione *ManageNewRequest*, che rappresenta il cuore del modulo server, verrà analizzato in seguito.

Come per la ricezione, anche l'invio della risposta continua fino a quando la funzione *NextSendResponse* rileva che tutti i dati sono stati inseriti nello stream di uscita. Successivamente la funzione attende che i dati vengano inviati correttamente, quindi chiude il socket definitivamente.

Il codice per la realizzazione del modulo server è abbastanza semplice, in quanto il framework .NET permette di programmare ad alto livello senza dover gestire tutte le pratiche relative alle connessioni TCP. Inoltre la struttura appena presentata permette di realizzare un server in grado di supportare più connessioni simultanee, limitando quindi eventi di sovraccarico che comprometterebbero l'affidabilità del sistema.

4.5.2 Esempio: scrittura dei programmi di cottura

Il vero cuore del modulo server, come è già stato detto, risiede nella funzione *ManageNewRequest*, che a differenza delle altre è l'unica che implementa un codice di interpretazione e generazione delle attività così come sono state presentate alle sezioni precedenti. Tutte le altre funzioni infatti sono preposte alla semplice gestione dei dati intesi come *flusso di byte*.

Vediamo quindi com'è strutturata la funzione e a titolo esemplificativo quali politiche vengono adottate nella gestione dell'attività di *scrittura dei programmi di cottura*.

La funzione inizia ricevendo in ingresso il socket e l'oggetto *StringBuilder* contenente tutti i dati ricevuti, che a questo livello vengono interpretati come una stringa di testo (la quale, in realtà, rappresenta l'XML di richiesta). La stringa viene quindi mappata in un oggetto di tipo *DataSet*; se tale conversione non dovesse riuscire (ad esempio perché i dati non sono stati ricevuti correttamente) la funzione abortisce e la richiesta viene ignorata.

A questo punto inizia la parte di interpretazione dell'XML e per prima cosa viene verificato il nome del DataSet con un costrutto di tipo *switch*:

```
1  switch (req.DataSetName)
2  {
3    case "getChildren":
4      [...]
5    break;
6    case "setProgrCott":
7      [...]
8    break;
9
10   /* Blocchi case relativi alle altre richieste */
11
12   default:
```

```
13      socket.Shutdown(SocketShutdown.Both);
14      socket.Close();
15    break;
16  }
```

Come è stato osservato alla sezione 4.4, ogni XML contiene come primo tag il nome dedicato alla richiesta, pertanto nella mappatura C# del DataSet il nome del tag verrà copiato nel nome del DataSet stesso.

In base al tipo di richiesta la funzione dovrà svolgere determinate funzioni e nel caso della scrittura dei programmi di cottura le operazioni da compiere sono le seguenti:

- la chiave relativa all'attività viene prelevata dal campo *key* della tabella *request* del DataTable
- viene verificato se la chiave è già presente nell'elenco delle attività da eseguire
  - se l'attività è presente, viene verificato se è stata completata (indifferentemente con esito positivo o negativo)
    - se l'attività è stata completata genera l'XML di risposta di tipo *setProgrCottT* con l'esito dell'operazione
    - se l'attività non è stata completata genera l'XML di risposta di tipo *busy*
  - se l'attività non è presente, verifica se contiene almeno un programma di cottura
    - se non ne contiene, genera l'XML di risposta di tipo *setProgrCottT* con l'esito dell'operazione (di default positivo)
    - se ne contiene, invoca la funzione *AddActivity* e genera l'XML di tipo *busy*

Il flusso di gestione dell'attività dovrebbe risultare abbastanza chiaro, poiché rispecchia le specifiche descritte alle sezioni precedenti. L'unica parte di codice che non è ancora stata presentata riguarda la funzione *AddActivity*, la quale si occupa di inserire nel database di appoggio l'istanza di una nuova attività ricevuta dal nodo padre (con le relative sotto-attività).

Come per la funzione *ManageNewRequest*, anche la funzione *AddActivity* include un grande blocco di tipo *switch* per identificare il tipo di attività che deve essere elaborata.

Nel caso della scrittura dei programmi di cottura il pseudo-codice seguente riassume quanto svolto dalla funzione:

```
1 Inserisci nella tabella activity del database una nuova riga con l'
    attività ricevuta
2 level = numero livello attività + 1;
3
4 // Generazione sotto-attività
```

```
5  Per ogni programma di cottura inserito nell'attività ricevuta
6  {
7    nodo = nodo destinatario del programma di cottura
8    if (nodo non è presente nel proprio sotto-albero)
9      salta il ciclo
10   if (nodo è di tipo nDir)
11     nodo = TopRef(nodo)
12   if (è già stata inserita una sotto-attività riferita all'attività
         madre per nodo)
13     salta il ciclo
14   keySubAct = GeneraNuovaChiaveUnivoca()
15   elencoProgrCott = Copy(elenco progr. cott. attività madre)
16   elencoFigli = Figli(nodo)
17   elencoFigli = elencoFigli + nodo
18   Per ogni programma di cottura in elencoProgrCott
19     Se il destinatario del programma non è in elencoFigli
20       Elimina programma da elencoProgrCott
21   xmlSubAct = GeneraXMLSetProgrCott(keySubAct, level, elencoProgrCott)
22   Inserisci nella tabella activitySplit del database una nuova riga con
         (keySubAct, nodo (destinatario), xmlSubAct)
23 }
```

In definitiva, "l'intelligenza" vera e propria di tutto il modulo server è racchiusa nelle funzioni *ManageNewRequest* e *AddActivity* che si occupano rispettivamente di gestire il flusso di gestione delle risposte e le routine di inserimento dell'attività madre e delle sotto-attività (instradamento delle richieste).

Chiaramente queste funzioni, con i metodi annessi, sono le uniche parti del codice che devono essere modificate per implementare le future procedure di gestione di nuove attività.

4.6 Modulo client (multi-thread)

La struttura del modulo client si differenzia da quella del modulo server per essere meno complessa nella parte di gestione delle comunicazioni, ma più complessa nelle politiche di gestione delle attività.

Innanzitutto va detto che il modulo client si divide in due parti: la prima inclusa nel "programma principale" e la seconda inserita in un'apposita classe. Iniziamo concentrandoci sulla prima parte.

Prima di continuare va detto che parlare di "modulo client" non è corretto, o meglio lascia intendere che il servizio di sistema sia provvisto di un codice in grado di gestire una sola richiesta in simultanea. URC è invece provvisto di un modulo multi-client, nel quale il numero di attività svolte in contemporanea può essere configurato a piacere in base alle capacità di calcolo del server in cui viene installato.

In sostanza il programma dispone di una serie di *client-slot* i quali possono essere assegnati in runtime ad attività differenti, prelevate direttamente dalla tabella *activitySplit* del database.

Ogni client-slot dispone di una "copia" del modulo client vero e proprio, il quale viene avviato in un thread dedicato. Quindi, riassumendo, supponendo di aver configurato 10 client-slot, il programma farà funzionare 10 moduli client indipendenti su

10 thread distinti, ognuno dei quali si occuperà di eseguire una specifica sotto-attività fra quelle presenti nel database di sistema.

In C# un client-slot è una struttura dati di questo tipo:

```
1 private struct ClientSlot
2 {
3   public Thread client;
4   public UNOXRemoteControlClient clientObj;
5   public string activityKey;
6   public string activityType;
7   public int timeout;
8 }
```

La struttura include il riferimento all'oggetto *Thread,* all'oggetto client *clientObj* assegnato al thread, alla *chiave*, al *tipo* e al *timeout* dell'attività scelta; va evidenziato che il timeout contenuto nella struttura non fa riferimento ai timeout presenti nelle tabelle tabelle *activity* e *activitySplit* del database. Se un client-slot è libero, l'oggetto *client* è uguale a *null*.

A questo punto è facile comprendere che si rileva necessario disporre di un qualche automatismo che si occupi di assegnare le sotto-attività ai client-slot disponibili; tale attività non dovrà essere eseguita in modo continuo, in quanto si corre il rischio di sovraccaricare di richieste i nodi presenti ai livelli inferiori. Diciamo quindi che la periodicità con cui tale attività viene svolta deve garantire una buona reattività del sistema (anche in sistemi con molti livelli), senza però causare situazioni di dropout dovute al diffondersi di troppe richieste simultanee.

Per rispondere a tali esigenze si è deciso di fare in modo che l'attività venga regolata da un *timer*, con cadenza configurabile dall'amministratore di sistema e impostata di default a 500 ms.

Quindi, per avviare il modulo client del servizio di sistema, come si è già visto nel caso del modulo server, il codice da inserire nel metodo *OnStart* del servizio di sistema è il seguente:

```
1 urcClientIncTick = 0;
2 urcClient = new Timer(new TimerCallback(ClientTimer));
3 urcClient.Change(100, urcClientInterval);
4 urcClientSlots = new ClientSlot[urcClientSlotCount];
```

In sostanza l'interprete .NET avvia un nuovo timer (oggetto *urcClient*) dopo una pausa iniziale di 100 ms e con cadenza impostabile a piacere (di default 500 ms); il timer ad ogni tick provvederà ad avviare la funzione di nome *ClientTimer*. Nella riga di codice numero 4 viene inizializzato l'array di strutture *ClientSlot*, il quale inizialmente presenterà tutti gli slot liberi. Il significato della prima riga di codice verrà invece chiarito in seguito.

La funzione *ClientTimer* è composta dunque dal codice che si occupa di gestire l'esecuzione delle attività. Il contenuto della funzione è il seguente:

```
1  TimerTickPerSecond = integer(1000 / urcClientInterval)
2  urcClientIncTick = urcClientIncTick + 1
3
4  if (urcClientIncTick % TimerTickPerSecond = 0)
5  {
6    Decrementa il timeout di ogni attività della tabella activity
7    Per ogni riga in activity con timeout = 0
8      Per ogni sotto-attività in activitySplit imposta lo status a error
9  }
10
11 if (urcClientIncTick % (TimerTickPerSecond * 60) = 0)
12   Rimuovi le righe in activity precedenti alle ultime 24 ore
13
14 Per ogni client slot
15 {
16   if (client slot non è occupato)
17     Salta il ciclo
18   slot.timeout = slot.timeout - 1
19   if (slot.timeout = 0)
20   {
21     Decrementa il timeout di activitySplit con key = slot.activityKey
22     if (activitySplit con key = slot.activityKey ha timeout = 0)
23     {
24       Imposta lo status di activitySplit a error
25       Abortisci il client dello slot
26       slot.client = NULL
27     }
28   }
29 }
30
31 urcClientSlotCountLow = urcClientSlotCount / 2
32 elencoAct = Preleva le righe di activitySplit con status = busy
33 Per ogni sotto-attività in elencoAct
34 {
35   subActType = tipo sotto-attività (nome della sotto-attività)
36   Cerca uno slot libero (se l'attività è a bassa priorità cerca solo
       nei primi urcClientSlotCountLow slot)
37   if (c'è uno slot libero)
38   {
39     if (sotto-attività è già stata attribuita ad un slot)
40       Salta il ciclo
41     Carica i dati nello slot con slot.timeout = SecondiTimeoutActType(
         subActType) * TimerTickPerSecond
42     Avvia il thread slot.client
43   }
44 }
```

Riassumendo, la funzione appena analizzata è costituita da 4 blocchi di codice distinti, i quali svolgono le seguenti attività:

- *ogni secondo* viene aggiornato il valore di *timeout* delle attività della tabella *activity* del database;
- *ogni 60 secondi* vengono eliminate dal database le attività più vecchie di 24 ore;

- ogni client-slot prevede un timeout interno, (espresso in numero di tick massimali del timer - variabile *timeout* della struttura *ClientSlot*) che limita ad un certo numero di secondi l'attività del thread di gestione del modulo client; se il timeout arriva a zero prima che l'attività del thread sia terminata, la funzione abortisce il thread (e di conseguenza anche il timer) e decrementa il valore del campo *timeout* in *activitySplit* (con il significato che un altro tentativo è andato fallito);

- le sotto-attività con *status* pari a *busy* (e che non compaiono negli slot occupati) vengono assegnate ad un client-slot libero; in questa fase il timeout relativo al client-slot viene impostato in base al tipo di attività da svolgere (ad esempio l'attività di aggiornamento del firmware avrà un timeout maggiore rispetto all'attività di lettura dei programmi di cottura, i quanto la mole di dati da trasferire è sicuramente maggiore).

Come si evince dal codice appena analizzato, tale funzione realizza tutte le regole di gestione dei timeout e delle politiche di priorità discusse alla sezione 4.2.3. Pertanto se in futuro vi fosse la necessità di modificare tali meccanismi basterà rivedere le parti di codice appena discusse.

Ciò che rimane ora da analizzare è la classe *UNOXRemoteControlClient*, ovvero il codice che si occupa di gestire le comunicazioni TCP e l'interpretazione degli XML inviati dai moduli server in risposta alle interrogazioni effettuate dal client.

4.6.1 Schema a blocchi

Come è già stato anticipato in precedenza, la struttura preposta alla gestione delle comunicazioni del modulo client risulta molto più semplice di quella del modulo server, in quanto, considerando che ogni modulo client viene eseguito in un apposito thread dedicato, le comunicazioni possono essere gestite utilizzando funzioni sincrone (o bloccanti).

Pertanto il modulo client dispone di un solo metodo per la gestione delle comunicazioni, il quale segue il flusso rappresentato in Figura 4.4.

Il metodo, dopo essere stato avviato attraverso l'esecuzione del relativo thread, preleva da una variabile oggetto dedicata la chiave della sotto-attività da eseguire; tale chiave viene impostata dalla funzione ClientTimer ,vista in precedenza, prima di avviare il thread. In Figura 4.4 la chiave in questione è quella dell'attività ACT.

Similmente a quanto avviene nel caso del modulo server, l'intera intelligenza viene mascherata nella funzione *ManageResponse*, la quale riceve in ingresso il listato XML di risposta per aggiornare i dati contenuti nel database MySQL.

4.6.2 Esempio: lettura dei programmi di cottura

In questa sezione analizziamo il lavoro svolto dalla funzione *ManageResponse* nel caso di lettura dei programmi di cottura, ovvero quando il server remoto restituisce un DataSet di nome *getProgrCottT*[2].

[2]si parla di DataSet per fare riferimento all'XML di mappatura dei dati contenuti nel DataSet stesso.

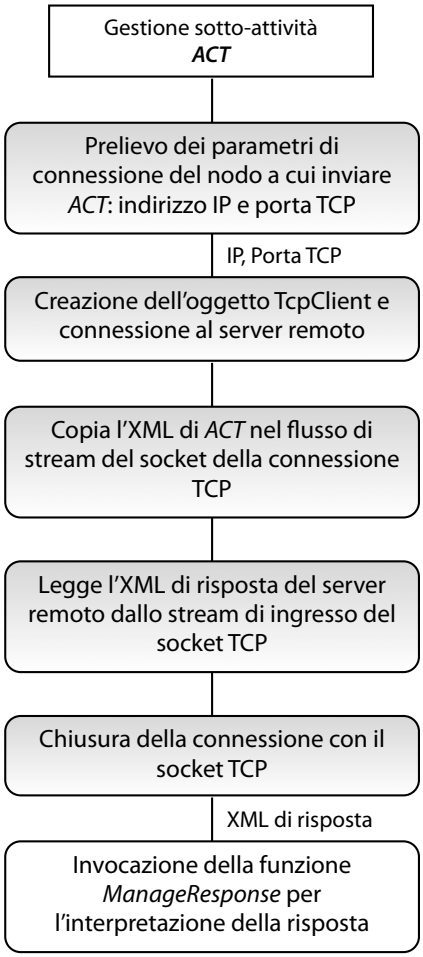

Figura 4.4: Schema a blocchi del modulo client del servizio di sistema.

Come si può ipotizzare, anche questa funzione include un grande blocco *switch* che seleziona il tipo di attività. Nel caso di lettura dei programmi di cottura, fra le direttive case-break del costrutto switch troviamo il seguente codice:

```
1  resultAct = xml di risposta
2  if (status di resultAct = complete)
3  {
4    Per ogni progrCott contenuto in resultAct
5    {
6      nodo = nodo che memorizza progrCott
7      progrCottNodo = estrai dalla tabella progrCott del database i
           programmi di nodo
8      if (progrCott è presente in progrCottNodo)
9        Aggiorna il programma di cottura presente nella tabella progrCott
             del database
10     else
11       Inserisci nuovo programma di cottura nella tabella progrCott del
             database
```

```
12    }
13 }
```

Le operazioni svolte dalla funzione sono dunque molto semplici e si traducono in semplici prelievi e aggiornamento dei dati presenti nel database in base alle informazioni ricevute dal nodo server di livello inferiore.

4.7 Libreria esterna

Il servizio di sistema, oltre ad essere dotato del codice fin quì analizato, utilizza un'apposita DLL esterna per svolgere alcune funzioni.

La DLL, chiamata *URCLibrary*, è composta da una serie di classi, prima fra tutte la classe *URCDatabase* (e relativa *URCDatabaseResult*) vista in precedenza.

La libreria ha lo scopo di raggruppare tutte le routine che sono richieste in diversi punti del programma principale, in modo tale che successive revisioni del codice possano essere svolte velocemente modificando solo i metodi della libreria.

L'elenco delle classi è il seguente:

- *URCPreferences*: contiene le funzioni dedicate al prelievo delle preferenze di sistema memorizzate nell'apposita tabella del database.
- *URCData*: prevede una serie di funzioni, tutte dedicate alla lettura/scrittura di dati nel database, come ad esempio:
  - *GetNodes(bool onlyDirect)*: preleva l'elenco di tutti i nodi presenti nella tabella *systemTree* del database. Se *onlyDirect* è true vengono prelevati solo i nodi diretti.
  - *GetNode(string name)*: preleva il nodo con nome *name* dalla tabella *systemTree*.
  - *GetActivityId(string key)*: restituisce l'id dell'attività con chiave specificata.
  - *SetActivityStatus(string key, string status)*: cambia lo stato di una sotto-attività.
  - *ActivityTimeout(string key)*: decrementa il timeout di una sotto-attività e, se necessario, ne aggiorna lo stato (se *timeout* vale zero la colonna status viene posta ad *error*).
  - *ActivityState(string key)*: restituisce lo stato di una attività sulla base degli stati delle relative sotto-attività.
- *URCAccessories*: contiene una serie di routine di carattere accessorio, come ad esempio:
  - *GetActivityKey(string nodeName)*: genera una nuova chiave per una attività.

  - *TimerTickPerSecond(int timerPeriod)*: ricava il numero di tick del timer del modulo client necessari a coprire un periodo di 1 secondo.

- *URCLog*: contiene una serie di routine per la generazione dei file di log del programma. Il servizio di sistema utilizza queste funzioni per tenere traccia di eventuali problemi (ad esempio all'avvio di eccezioni C#), cosicché possano essere identificati anche in caso di arresto imprevisto del programma.

Capitolo 5

L'interfaccia utente web-based

L'interfaccia è la parte del sistema a diretto contatto con l'utente. Perché il sistema risulti semplice da utilizzare agli occhi degli utilizzatori, occorre che il layout e l'organizzazione degli elementi interni alle maschere di inserimento dati sia ordinata ed intuitiva. Questo implica che l'interfaccia, prima di essere realizzata, debba essere analizzata e studiata nel dettaglio: anche i particolari più piccoli possono fare la differenza in termini di apprezzamento da parte degli utenti.

L'interfaccia rappresenta quindi il mezzo attraverso il quale l'utente compie il proprio lavoro e proprio per tale motivo, nella maggior parte dei casi, i sistemi informatici vengono valutati sulla base di questi parametri; l'intera logica di funzionamento del programma viene nascosta dalle maschere di interfaccia e pertanto occorre che queste siano affidabili e ben strutturate tanto quanto le routine di gestione dei dati.

Per i motivi appena esposti, in URC lo studio dell'interfaccia, che come abbiamo visto è di tipo web-based, offronta problematiche legate all'usabilità e si propone di risolverle attraverso l'adozione delle migliori tecnologie attualmente disponibili in ambito Web.

5.1 Browser web, server web e HTML

L'analisi svolta alla sezione 3.3 ha stabilito che l'interfaccia utente più idonea per il sistema URC sia di tipo *web-based*.

Più precisamente l'espressione web-based è impiegata nell'ambito del software engineering, dove con il termine *webapp* si descrive un'applicazione accessibile via web per mezzo di un network, come ad esempio una intranet, o attraverso la Rete Internet.

Questo modello applicativo è divenuto piuttosto popolare alla fine degli anni novanta, in considerazione della possibilità per un client generico di accedere a funzioni applicative, utilizzando come terminale normali web browser. Infatti l'opportunità di aggiornare ed evolvere a costo ridotto il proprio applicativo, senza essere costretti a distribuire numerosi aggiornamenti ai propri clienti attraverso supporti fisici, ha reso la soluzione piuttosto popolare per molti produttori software. Più di recente colossi come Google e Microsoft hanno implementato interi pacchetti applicativi per

office, tradizionalmente venduti in modo distribuito su supporti CD-ROM, e che ora si stanno velocemente trasformando a tutti gli effetti in webapps.

Ciò che è importante capire è che sviluppare applicazioni web-based significa aderire agli standard utilizzati in ambito Internet. Più precisamente occorre fare una grande distinzione fra due tipi di linguaggi, i quali si dividono in base all'interprete che si occupa della loro esecuzione: linguaggi *server-side* e linguaggi *client-side*.

La rete Internet, infatti, basa il proprio funzionamento sul paradigma client-server, già analizzato a lungo nella trattazione svolta alla sezione precedente. Più precisamente quando si digita un qualsiasi indirizzo su un comune *browser web*, in realtà, si genera una richiesta (solitamente utilizzando il protocollo *HTTP*) verso un *server web* remoto. Il server, una volta ricevuta la richiesta con l'indicazione della pagina che deve essere prelevata (parametri GET), provvede a trasmetterla al browser, il quale la visualizzerà sullo schermo dell'utente.

Questo meccanismo, che così rissunto potrebbe sembrare alquanto semplice, nasconde in realtà molti processi di elaborazione e parsing dei documenti, i quali si concentrano però in due fasi precise:

- lato server, nella restituzione della pagina richiesta;
- lato client, nella visualizzazione della pagina all'utente.

In queste due fasi infatti, server web da un lato e browser dall'altro, prima di restituire il risultato, interpretano il contenuto delle pagine, le quali possono includere script atti a modificarne il contenuto. Per tale motivo i linguaggi di realizzazione di tali script saranno di tipo client-side, se vengono interpretati dal browser, mentre saranno di tipo server-side, se vengono interpretati dal server.

In ambito web tutte le interfacce, che siano queste dei siti, o delle webapp, vengono disegnate utilizzando un particolare linguaggio: l'HTML.

L'HTML non è un linguaggio di programmazione, ma un linguaggio di markup, ossia descrive le modalità di impaginazione, formattazione, o visualizzazione grafica (layout) del contenuto, testuale e non, di una pagina web. Tuttavia, l'HTML supporta l'inserimento di script e oggetti esterni quali immagini o filmati.

L'HTML è un linguaggio di pubblico dominio la cui sintassi è stabilita dal World Wide Web Consortium (W3C) e che è basato su un altro linguaggio avente scopi più generici, l'SGML. È stato sviluppato alla fine degli anni ottanta da Tim Berners-Lee al CERN di Ginevra assieme al noto protocollo HTTP, che supporta invece il trasferimento di documenti in tale formato.

Di per sè l'HTML permette solo di disegnare la pagina e di inserirne i contenuti, che sono però di tipo *statico*. Questo implica che tutto quello che compone la pagina viene inserito in un documento HTML, il quale viene richiesto (dal browser) al server in seguito ad una richiesta HTTP. Ma se i contenuti della pagina devono poter essere modificati per presentare dei dati *dinamici*? Se, come nel nostro caso, volessimo ad esempio presentare la lista dei nodi server collegati al sistema, la quale risiede nel database e viene aggiornata con una certa regolarità?

Per rispondere a questi quesiti in sostanza si deve poter disporre di linguaggi di programmazione che siano in grado di alterare il codice HTML sulla base di una serie

di condizioni o di dati che si hanno a disposizione, in modo che i dati presentati non siano solo statici, ma possano essere arricchiti da contenuti dinamici.

Proprio per tale ragione, negli anni, in ambito Web, sono nati diversi linguaggi di programmazione, i quali nello specifico svolgono le seguenti funzioni.

5.1.1 Linguaggi lato server

I linguaggi di programmazione lato server vengono interpretati dal server web prima che il codice HTML che compone la pagina richiesta venga restituito al browser. In sostanza questi linguaggi vengono inseriti nelle pagine assieme al codice HTML e il loro flusso interpretativo modifica l'output del codice HTML finale.

Il codice dei linguaggi lato server, una volta inserito nelle pagine web, viene identificato dal server attraverso degli opportuni tag marcatori; pertanto il codice contenuto all'interno dei marcatori viene interpretato dal server, mentre tutto il codice esterno (ovvero il codice HTML statico) viene restituito al browser senza subire alterazioni di alcun genere.

I linguaggi lato server più diffusi sono il *PHP* e l'*ASP*. Mentre il primo è supportato da qualsiasi tipo di piattaforma, il secondo funziona solo su server Microsoft. In base al linguaggio scelto si dovrà fare riferimento ad un particolare web server, il quale contiene il motore di interpretazione del codice. ASP, essendo un linguaggio proprietario, viene interpretato solo da server IIS di Microsoft, mentre PHP, essendo un linguaggio completamente open source, viene interpretato da diversi web server. Il web server più utilizzato in abbinata al PHP è Apache, sviluppato dalla Apache Software Foundation.

5.1.2 Linguaggi lato client

I linguaggi di programmazione client-side sono tutti quei linguaggi di scripting che vengono interpretati ed eseguiti dal browser dopo che la pagina web viene ricevuta dal server web (si evidenzia che la pagina in questione può avere già subito un processo di manipolazione lato server attraverso linguaggi come PHP o ASP).

Gli script lato server, che vengono trasmessi assieme al codice HTML della pagina, sono dedicati alla manipolazione runtime degli elementi che compongono il layout e i contenuti della pagina stessa. L'aspetto più importante è che, a meno di caricamento di nuovi contenuti, tali manipolazioni possono essere svolte senza dover effettuare nuove richieste HTTP al server, poiché sarà direttamente il browser web ad occuparsi di effettuare le trasformazioni. Ad esempio, alla pressione di un semplice link HTML, attraverso una semplice riga di codice, è possibile modificare il colore di una determinata parte di testo.

Negli ultimi anni il linguaggio di programmazione lato client più utilizzato e diffuso anche nelle webapp più imponenti (come Google Maps, Google Docs, ecc) è *JavaScript*. Fu originariamente sviluppato da Brendan Eich della Netscape Communications con il nome di Mocha e successivamente di LiveScript, ma in seguito è stato rinominato "JavaScript" ed è stato formalizzato con una sintassi più vicina a quella del linguaggio Java di Sun Microsystems. JavaScript è stato standardizzato per la prima volta tra il 1997 e il 1999 dalla ECMA con il nome ECMAScript. L'ultimo

standard, del dicembre 1999, è ECMA-262 Edition 3 e corrisponde a JavaScript 1.5. È anche uno standard ISO.

5.2 L'ambiente server-side

In questa sezione del testo analizzeremo le tecnologie e i sistemi server-side che sono stati scelti per URC per realizzare un'applicativo di interfaccia utente dinamico e che sia in grado di soddisfare le esigenze fin qui analizzate.

Per prima cosa, facendo riferimento anche alle specifiche di sistema riportate alla sezione 2.3, risulta opportuno fare un elenco delle caratteristiche che dovranno essere incluse nell'applicativo di interfaccia utente:

- *presentazione dinamica dei dati*: i dati e gli elementi visualizzati nell'applicativo di interfaccia devono poter essere caricati dinamicamente, attraverso la lettura delle tabelle del database MySQL; l'applicativo deve essere in grado anche di inserire nuovi dati nel database e modificare quelli esistenti, ad esempio per il setup delle preferenze di sistema, o per l'avvio di nuove attività;

- *gestione multilingua*: tutti i testi presentati nella webapp di interfaccia devono poter essere localizzati in più lingue;

- *gestione multi-utente*: l'interfaccia deve poter essere utilizzata da utenti differenti per scopi differenti; pertanto è importante che l'accesso a determinate funzionalità del programma non venga consentito a chiunque;

- *elevate performance*: è importante che l'interfaccia utente sia snella e che le reazioni agli input degli utenti vengano gestite in modo rapido, in prospettiva di un utilizzo del sistema in mobilità (che come ambito di per sé può causare dei rallentamenti).

Ad un'analisi sommaria possiamo già dire che i primi tre requisiti necessitano sicuramente di una manipolazione del codice HTML e pertanto occorre fare in modo che l'ambiente di interfaccia lato server permetta l'utilizzo di un qualche linguaggio di programmazione. Si è già visto alla sezione 3.4 che la struttura del sistema prevede l'installazione presso ogni nodo server di un server web in grado di caricare l'applicativo di interfaccia. Basterà quindi decidere a quale web server affidarsi e soprattutto quale linguaggio di scripting utilizzare.

5.2.1 Il linguaggio PHP e la connessione al database MySQL

Il linguaggio server-side scelto per implementare la webapp di interfaccia del sistema URC è il PHP.

PHP riprende per molti versi la sintassi del C, come peraltro fanno molti linguaggi moderni, e del Perl. È un linguaggio a tipizzazione debole e dalla versione 5 migliora il supporto al paradigma di programmazione ad oggetti. Certi costrutti derivati dal C, come gli operatori fra bit e la gestione di stringhe come array, permettono in alcuni casi di agire a basso livello; tuttavia è fondamentalmente un linguaggio di

alto livello, caratteristica questa rafforzata dall'esistenza delle sue moltissime API, oltre 3.000 funzioni del nucleo base. PHP è in grado di interfacciarsi a innumerevoli database, tra cui MySQL.

Proprio per queste ragioni e per la semplice sintassi che lo contraddistingue PHP viene oggi scelto da molti sviluppatori web che in quest'ambito vedono nell'utilizzo congiunto con MySQL un punto di forza.

Per quanto riguarda URC, il PHP è stato scelto anche per offrire una maggiore compatibilità del sistema, considerando che il linguaggio viene supportato da diversi web server disponibili per la maggior parte delle piattaforme presenti in commercio.

Ma l'aspetto più importante del linguaggio è la moltitudine di API disponibili, le quali permettono di coprire qualsiasi tipo di esigenza. E in quest'ambito il PHP mette in risalto le sue capacità nell'interfacciamento a MySQL, e nel caso di URC questa è un'esigenza primaria.

Il suo maggior avversario, il linguaggio ASP (o ASP.NET) di Microsoft, seppur ben strutturato, presenta le seguenti problematiche, che nel caso di URC possono rappresentare un grosso limite:

- è supportato solo da sistemi Microsoft (server IIS);
- è un linguaggio proprietario (chiuso), per cui le API e le funzionalità vengono rilasciate solo da Microsoft;
- è nato per l'interfacciamento a database di casa Microsoft, come Access o Microsoft SQL Server, per cui il supporto a database MySQL non viene gestito in modo ottimale.

Per cogliere alcuni aspetti principali del linguaggio PHP si riportano ora alcuni esempi di codice.

Prima di tutto va detto che ogni routine PHP inizia con i simboli "<?" o "<?php" e termina con i simboli "?>". Per cui all'interno del codice HTML gli script PHP possono essere risconosciuti molto velocemente. Altro aspetto fondamentale è l'uso delle variabili, che in PHP non sono tipizzate (non si fa differenza fra variabili di tipo integer o di tipo stringa) e che presentano un nome che viene sempre preceduto dal simbolo "$".

Lo script seguente, ad esempio, permette di stampare una tabella HTML contenente l'elenco dei nodi di tipo *device* del sistema:

```
1  <html>
2  <body>
3
4  <?php
5  $systemTree = array();
6
7  /* Codice per la connessione al database MySQL e l'esecuzione della
       query "SELECT name FROM systemTree WHERE type='device'" per il
       caricamento dei dati nel DataTable systemTree */
8
9  echo '<table>';
10 echo '<tr><td>Id</td><td>Nome</td></tr>';
```

```
11 foreach ($systemTree as $id => $value)
12   echo '<tr><td>'.$id.'</td><td>'.$value.'</td></tr>';
13 echo '</table>';
14 ?>
15
16 </body>
17 </html>
```

L'esempio evidenzia come codice HTML e codice PHP coesistano all'interno dello stesso documento e come, in modo simile a quanto accade con C#, i dati possano essere manipolati in modo semplice e facendo uso di poche righe di codice. Nell'esempio, infatti, gli elementi dell'array *$systemTree* vengono stampati ciclicamente e strutturati in un elemento *<table>*.

L'esempio seguente, invece, completa il codice precedente aggiungendo le routine di connessione al database e prelevando i dati dalla tabella *systemTree*:

```
1  <html>
2  <body>
3
4  <?php
5  $systemTree = array();
6  // Connessione al database
7  $mysqli = new mysqli("localhost", "root", "root", "urc", "42110");
8  $stmt = $mysqli->stmt_init();
9  if ($stmt = $mysqli->prepare("SELECT␣name␣FROM␣systemTree␣WHERE␣type=?"
       ))
10 {
11   $stmt->bind_param("s", "device");
12   $stmt->execute();
13   $stmt->bind_result($name);
14   $ind = 0;
15   while($stmt->fetch())
16     $systemTree[$ind++] = $name;
17   $stmt->free_result();
18   $stmt->close();
19 }
20 $mysqli->close();
21
22 echo '<table>';
23 echo '<tr><td>Id</td><td>Nome</td></tr>';
24 foreach ($systemTree as $id => $value)
25   echo '<tr><td>'.$id.'</td><td>'.$value.'</td></tr>';
26 echo '</table>';
27 ?>
28
29 </body>
30 </html>
```

In sostanza la classe preposta a stabilire una connessione con il database MySQL è *mysqli* (in realtà le API dedicate a tale scopo in PHP sono molteplici, ma la più performante e di ultima generazione è proprio mysqli). Invece l'oggetto *$stmt*

permette di svolgere lo stesso compito che in C# viene delegato alla classe *MysqlCommand*. L'oggetto permette in sostanza di preparare delle query SQL complete di parametri da mandare in esecuzione, facendo in modo che il programmatore non si debba preoccupare di gestire tutti gli aspetti connessi all'attività, come ad esempio il mapping corretto dei caratteri speciali.

5.2.2 Il server web Apache e il ModRewrite per il multilingua

Apache è il web server più diffuso al mondo. Compatibile con sistemi Unix-Linux e Microsoft (oltre agli altri), Apache supporta nativamente l'interpretazione di script PHP. Per tali ragioni e per la sua nota fama è stato scelto come modulo web server nel sistema URC.

Il web server Apache presenta un'architettura modulare, quindi ad ogni richiesta del client vengono svolte funzioni specifiche da ogni modulo di cui è composto, come unità indipendenti. Ciascun modulo si occupa di una funzionalità, ed il controllo è gestito dal core. Il motore di interpretazione di PHP è proprio un dei moduli di Apache.

Per configurare il web server gli amministratori di sistema possono usare il file *httpd.conf*. Questo file mette a disposizione tutta la libertà offerta dal server, quindi impostare la porta TCP di funzionamento del server, aggiungere moduli, estensioni, nuovi mime-type ed altro ancora. Per esempio, se si volesse aggiungere un modulo bisognerebbe usare questa sintassi: *LoadModule nome_modulo percorso_del_file*.

Nel sistema URC Apache viene utilizzato con la configurazione di default, pertanto, una volta creato un pacchetto di installazione ad-hoc, non è necessario che l'amministratore di sistema intervenga per modificare il file httpd.conf.

Un altro modo per configurare il funzionamento del server in modo più specifico è fare ricorso ai file *.htaccess*, i quali permettono un'ulteriore personalizzazione del web server a livello di directory. In sostanza in ogni directory del progetto è possibile inserire un file .htaccess (nei sistemi Unix-Linux il punto all'inizio del nome indica che il file è di tipo nascosto) che può contenere delle direttive di configurazione specifiche per quella determinata directory.

Il caso più diffuso è quello di usare questi file per gestire il comportamento di un particolare modulo di Apache: il ModRewrite. Come dice la pagina ufficiale del sito di Apache, "ModRewrite provides a rule-based rewriting engine to rewrite requested URLs on the fly", il server, prima di analizzare l'URL ricevuto nella richiesta HTTP, può riscriverlo applicando le regole contenute nel file di configurazione. Le regole sono basate sull'uso delle *espressioni regolari*.

Ecco un esempio che illustra un classico utilizzo del ModRewrite:

```
1 RewriteEngine On
2 RewriteRule ^en/(.+)\.php$ $1.php?lang=en [QSA]
3 RewriteRule ^ita/(.+)\.php$ $1.php?lang=ita [QSA,L]
```

Questo è proprio il file .htaccess presente nella root directory della webapp di URC. Queste direttive permettono di gestire il multilingua in modo "intelligente" e il loro significato è il seguente:

- Nella riga 1 il motore di Rewrite viene abilitato.
- La riga 2 serve per gestire la localizzazione in inglese. Quando Apache riceve un URL composto da "en/" più una qualsiasi stringa che termina con ".php" memorizza tutta la stringa nella variabile $1. Poi il secondo blocco della direttiva alla riga 2 dice ad Apache che l'URL così individuato va tradotto in $1 concatenato a ".php?lang=en". Il comando *[QSA]* stabilisce che eventuali parametri GET presenti dopo il nome della pagina *.php* individuata debbano essere aggiunti a quelli esistenti (cioè *lang*).
- La riga 3 funzione allo stesso modo, ma serve per gestire la lingua italiana. Il parametro *[L]* dice ad Apache che quella direttiva è l'ultima del file.

Ma perché in URC si utilizza il ModRewrite per gestire il multilingua? La risposta è semplice: per utilizzare dei file di script PHP univoci (un'unica pagina per tutte le lingue) ed evitare di dover inserire in ogni link (o URL) presente all'interno del programma il riferimento al parametro GET *lang*. In sostanza bisogna fare in modo che tutti gli script PHP possano recuperare l'indicazione con la lingua impostata dall'utente per visualizzare i testi dei contenuti nella lingua scelta. Chiaramente ci sono diversi modi per farlo ed alcune varianti consistono nell'utilizzare i Cookie o le sessioni PHP[3]. Ma il modo più semplice ed elegante per trasportare questo tipo di impostazione senza bisogno di scrivere ulteriore codice PHP, è usare il parametro GET in abbinata al ModRewrite. Vediamo subito perchè.

Ad esempio, consideriamo l'URL "en/setup/setup.php?comando=insert"; in base alle regole viste in precedenza (riga 2 dell'esempio) l'URL viene tradotto in "setup/setup.php?lang=en&comando=insert". In sostanza, come è facile intuire, l'indicazione della lingua viene prelevata dal percorso dell'URL ed inserita come parametro GET. Il vantaggio nell'uso di questo meccanismo sta nel fatto che tutti i link della webapp potranno far uso di URL relativi, al fine di trasportare implicitamente proprio nell'URL richiesto l'indicazione della lingua impostata (senza bisogno di ricorrere al parametro GET *lang* in modo esplicito). Quindi se è stata impostata la lingua inglese, stiamo visitando la pagina "en/index.php" e in questa pagina siamo in presenza di un link che punta alla pagina "setup/setup.php", nel percorso del link possiamo inserire l'URL relativo "./setup/setup.php". Il browser alla pressione del link, essendo l'URL di tipo relativo, punterà alla pagina "en/setup/setup.php", trasportando implicitamente anche l'impostazione della lingua. Chiaramente sia la pagina "en/index.php" che la pagina "en/setup/setup.php" nelle directory della webapp a livello fisico non esistono, poiché il prefisso "en/" è solo virtuale; a livello fisico esistono le pagine "index.php" e "setup/setup.php" che sono proprio quelle che vengono prelevate da Apache dopo aver applicato il ModRewrite. Tali pagine potranno recuperare la lingua impostata dall'utente leggendo il contenuto della variabile GET *lang*.

5.3 L'ambiente e le tecnologie client-side

Dopo aver presentato i linguaggi e le tecnologie utilizzate in ambito server, analizziamo quelle usate dalla webapp di URC in ambito client.

UNOX fin dalle prime fasi di sviluppo del sistema ha voluto fare in modo che l'applicativo di interfaccia venisse realizzato con le più moderne tecnologie utilizzate in ambito Web, seguendo lo stile imposto dall'ormai famoso Web 2.0.

Nel Web 2.0 molte delle applicazioni web che vengono sviluppate fanno uso di linguaggi lato client in modo tale da rendere le pagine dinamiche sotto il profilo della presentazione dei dati. Con il termine "dinamiche" non si intende che le pagine debbano contenere animazioni o effetti grafici particolari, ma che gli elementi e i contenuti presenti nella pagina possano essere modificati e aggiornati in tempo reale senza dover ricaricare di volta in volta la pagina stessa. Ad esempio, se l'utente preme un pulsante per inserire un nuovo programma di cottura, con il vecchio stile di programmazione vi erano due possibilità:

- apertura nella stessa finestra del browser di un'apposita pagina (HTML o PHP) contenente un form HTML;
- apertura della stessa pagina in una nuova finestra del browser.

La prima soluzione è senza dubbio da preferire alla seconda, in quanto l'apertura di nuove finestre aumenta lo stress dell'utente durante la sua esperienza di navigazione. Ma in ogni caso nemmeno la prima soluzione è ideale, in quanto l'apertura di una nuova pagina nella stessa finestra del browser comporta la chiusura dei contenuti precedentemente visualizzati.

Per evitare di incorrere in questi inconvenienti e per rendere l'applicazione più veloce ed intuitiva, ciò che può essere fatto è che il nuovo form di inserimento del programma di cottura compaia all'interno della stessa pagina, ad esempio tramite la comparsa di un nuovo apposito riquadro.

Per riuscire ad implementare tale meccanismo occorre però ricorrere a linguaggi di programmazione lato client, i quali permettono di manipolare gli elementi HTML presenti nella pagina.

Come vedremo in seguito, la webapp di URC fa largo uso di tutte queste tecniche, al fine di rendere l'applicazione più intuitiva e semplice da utilizzare.

5.3.1 I fogli di stile CSS

Un aspetto presente in tutte le webapp di ultima generazione (e più in generale in tutto il Web moderno) è l'uso dei fogli di stile CSS (Cascading Style Sheets). Fino a qualche anno fa, infatti, il layout, i colori e gli stili della pagina venivano inseriti direttamente nel codice HTML. Oggigiorno invece si preferisce fare in modo che il codice HTML definisca solo la struttura della pagina a livello di *blocchi di contenuti*, delegando ai CSS la parte di definizione dello stile (colori, font, bordi, margini, posizionamenti, ecc).

Le regole per comporre il CSS sono contenute in un insieme di direttive (Recommendations) emanate a partire dal 1996 dal W3C.

A titolo esemplificativo, un estratto di foglio di stile utilizzato dalla webapp di URC è il seguente:

```
1 /***** GENERALI *****/
2 body { background-color:#F9F9F9; text-align:left; padding:20px 0px 0px
      0px; font:12px verdana,sans-serif; color:#000000; }
3 body a { text-decoration:none; font:12px verdana,sans-serif; color
      :#0066FF; }
4 body a:hover { text-decoration:none; background-color:#CCDDFF; font:12
      px verdana,sans-serif; color:#0066FF; }
5 body a img { border:0; }
6 h2 { text-decoration:none; font:15px verdana,sans-serif; color:#000000;
       background:none; }
7 .mexOk { width:80%; margin:20px auto 20px auto; padding:5px; font:12px
      verdana,sans-serif; color:#336600; background-color:#EAFFDF; border
      :1px dashed #336600; text-align:center; display:none; }
8 .mexErr { width:80%; margin:20px auto 20px auto; padding:5px; font:12px
       verdana,sans-serif; color:#B74900; background-color:#FFDFB0; border
      :1px dashed #B74900; display:none; }
```

Nonostante vi sia quindi una standardizzazione delle regole, la più grande problematica connessa all'uso dei CSS sta nella compatibilità cross-browser.

Ciò significa che browser diversi possono interpretare il codice CSS in modi totalmente diversi; il risultato è che la maggior parte delle volte la pagina presenta delle problematiche di layout che rendono difficile (se non impossibile) l'interpretazione dei contenuti da parte dell'utente. L'unico modo per risolvere il problema è effettuare un'attenta analisi di debug con svariati test di compatibilità della webapp grazie a diversi browser durante tutta la fase di sviluppo dell'applicazione. Questa attività durante le fasi di sviluppo di URC è stata svolta pesantemente, per garantire una piena compatibilità dell'applicazione con tutti i più diffusi browser web nel pieno rispetto delle specifiche analizzate alla sezione 2.3 del testo.

5.3.2 JavaScript, jQuery e AJAX

Nella webapp di URC JavaScript viene usato in modo massiccio per la gestione dinamica di tuti i contenuti che compongono le pagine.

Anche JavaScript, però, un po' come avviene per i CSS, presenta alcune problematiche nella compatibilità cross-browser. Sebbene le caratteristiche del linguaggio siano ben definite, alcune funzioni possono avere esiti e comportamenti diversi in browser diversi o, a volte, addirittura in versioni diverse dello stesso browser.

Inoltre la scrittura di determinate funzioni, come la selezione di un elemento DIV[1] di una pagina HTML, possono richiedere più righe di codice, soprattutto in virtù di quanto appena detto (per la necessità di verificare la famiglia di browser che sta eseguendo lo script). Questa problematica si aggrava quando si vogliono applicare degli effetti grafici alle pagine.

Par tale motivo negli ultimi anni, complice inoltre la diffusione sempre più capillare di JavaScript all'interno delle pagine web, sono nati diversi framework (scritti per l'appunto in JavaScript) che permettono di risolvere tutte queste problematiche. Questi framework contengono in sostanza una serie di API che permettono di svol-

[1] l'elemento DIV consente di definire un riquadro di contenuti.

gere una gamma di funzioni scrivendo poche righe di codice. Il punto di forza di queste librerie è la piena compatibilità cross-browser di tutte le API.

Il framework più popolare, completo e performante è *jQuery*[5]. Tramite l'uso della libreria jQuery è possibile con poche righe di codice effettuare molte operazioni, come ad esempio ottenere l'altezza di un elemento, o farlo scomparire con effetto dissolvenza. Anche la gestione degli eventi è completamente standardizzata e gestita automaticamente assieme alla loro propagazione: è possibile ad esempio fare in modo che una determinata funzione JavaScript venga richiamata alla pressione di un tasto.

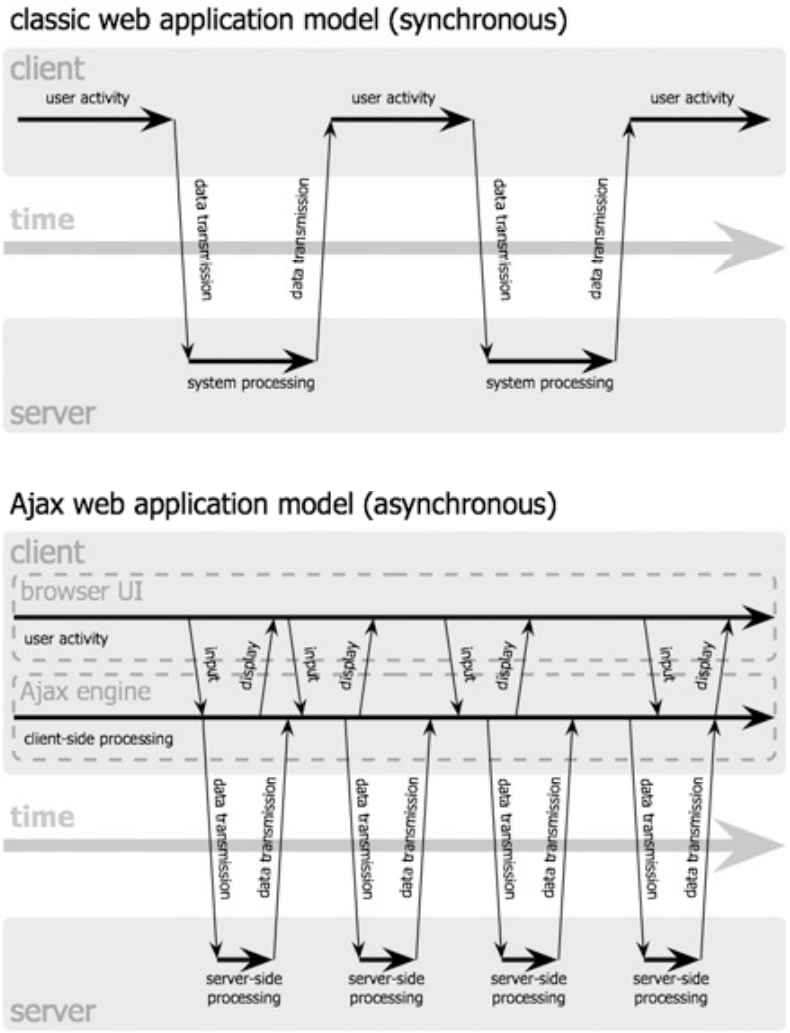

Figura 5.1: Interazione dei componenti AJAX paragonati alla staticità di un'applicazione web tradizionale.

L'oggetto principale di nome jQuery è genericamente utilizzato tramite il suo alias, il carattere *$*, per mantenere uniformità con la libreria Prototype (altro framework JavaScript).

I selettori in jQuery utilizzano la sintassi dei selettori CSS; sono quindi concatenabili e molto precisi nel ritornare gli elementi voluti.

Un altro vantaggio derivante dall'utilizzo di jQuery consiste nella moltitudine di progetti disponibili in rete (molto spesso con licenza Gnu GPL), i quali basano il proprio funzionamento proprio su jQuery. Pertanto aggiungere alla pagina elementi di interfaccia grafica avanzati, come strutture ad albero o menu a tendina, diventa davvero molto semplice.

Altra tecnologia che viene utilizzata in URC è *AJAX*[6]. AJAX, acronimo di Asynchronous JavaScript and XML. Questa è una tecnica di sviluppo per la realizzazione di applicazioni web interattive (Rich Internet Application). Lo sviluppo di applicazioni HTML con AJAX si basa su uno scambio di dati in *background* fra web browser e server, che consente l'aggiornamento dinamico di una pagina web senza esplicito ricaricamento da parte dell'utente (vedi Figura 5.1).

AJAX è asincrono nel senso che i dati extra sono richiesti al server e caricati in background senza interferire con il comportamento della pagina esistente. Normalmente le funzioni richiamate sono scritte con il linguaggio JavaScript. Tuttavia, e a dispetto del nome, l'uso di JavaScript e di XML non è obbligatorio, come non è necessario che le richieste di caricamento debbano essere necessariamente asincrone.

Ad ogni modo il vantaggio più grande di usare AJAX è la grande velocità con la quale un'applicazione risponde agli input dell'utente.

5.4 Studio dell'interfaccia utente

Lo studio dell'interfaccia utente del sistema URC si pone come obiettivo principale di portare l'usabilità dell'applicazione ai massimi livelli. Questo, nell'ottica di lavoro di UNOX richiede la massima semplicità.

Semplificare la struttura di interfaccia significa garantire elevate prestazioni (anche in mobilità), pulizia del layout e maggiore facilità d'uso. Per tale ragione, fin dalle fasi di progettazione della grafica, si è cercato di andare in tale direzione, sviluppando un'interfaccia semplificata sia sotto il profilo grafico, sia sotto quello delle funzionalità.

Per cominciare occorre fare un elenco delle funzionalità che dovranno essere implementate nell'applicativo di interfaccia:

- Gestione delle diverse attività: la funzione principale consiste nell'avere il controllo sulle attività svolte dal servizio di sistema; per tale ragione le maschere di interfaccia dovranno consentire il recupero e la modifica dei dati salvati nella tabella *activity* del database e in tutte le altre tabelle di appoggio.

- Setup del sistema: la webapp di interfaccia utente dovrà permettere una completa configurazione del sistema, ad esempio per la parte di preferenze e di configurazione dei figli diretti del nodo.

- Gestione utenti: considerando che la webapp può essere usata da utenti differenti con privilegi diversi risulta importante prevedere un sistema di autenticazione e modifica degli utenti abilitati ad accedere al sistema.

- Gestione multilingua: l'applicazione dovrà permettere la presentazione dei contenuti in più lingue (l'idea che sta alla base di tale funzionalità è già stata vista in precedenza).

Partendo dunque da tali considerazioni e cercando di aderire agli standard del settore, si è pensato di strutturare l'interfaccia come un classico pannello di amministrazione, nel quale le diverse funzionalità possono essere raggiunte attraverso un apposito menu di selezione.

La struttura del menu della webapp di URC è rappresentata in Figura 5.2.

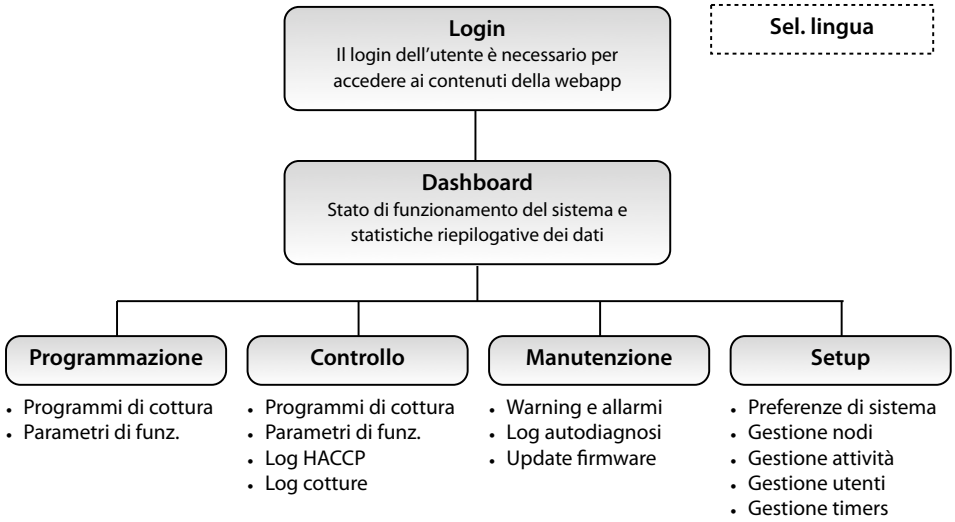

Figura 5.2: Struttura del menu funzioni dell'applicativo di interfaccia utente.

Come è visibile nell'immagine si è deciso che tutti i contenuti della webapp debbano essere raggiungibili previa autenticazione dell'utente; il login rappresenta quindi una condizione necessaria per poter utilizzare l'applicazione. Una volta effettuato il login si accede alla Dashboard, una pagina che riassume lo stato del sistema e una serie di dati statistici, come ad esempio il numero di nodi, il tempo medio di funzionamento dei forni collegati al sistema, ecc. Le diverse funzioni sono state invece raggruppate in 4 sezioni principali: programmazione, controllo, manutenzione e setup; ogni area racchiude in sè le attività appartenenti a quella specifica categoria. La selezione della lingua, invece, appare come una funzionalità del sistema alla quale si può accedere in qualsiasi momento, senza bisogno di accedere ad una particolare pagina.

Vediamo dunque come sono strutturati i contenuti di ogni diversa pagina analizzando il template grafico e il layout della webapp di URC.

5.4.1 Template grafico, layout e analisi di usabilità

Per la realizzazione del template grafico si è pensato di adottare uno stile sobrio, ma pur sempre elegante, e che segua le linee guida del più semplice layout in ambito web, composto dai blocchi header (testata), content (contenuto) e footer (fondo pagina).

Inoltre per il disegno della grafica si è pensato di mantenere il set di colori usato solitamente dall'azienda: nero preponderante e scale di grigi.

Figura 5.3: Template grafico dell'applicativo di interfaccia utente.

Per fare in modo che l'utente possa raggiungere in ogni momento tutte le funionalità del sistema si è deciso di posizionare nell'header della pagina un menu di selezione, il quale risulta ben visibile e facilmente accessibile. La posizione di tale menu non è stata scelta casualmente, ma deriva dalle regole di usabilità del mondo web. In quest'ambito, infatti, la parte della pagina che più di tutte attira l'attenzione dell'utente è quella superiore. Posizionando il menu in tale zona si soddisfano inoltre le esigenze orientative dell'utente, in modo tale che non si affatichi durante l'esperienza di navigazione (perché, ad esempio, non sa spiegarsi come raggiungere la pagina dedicata ad una determinata funzionalità).

Un'altra scelta fondamentale è stata quella di rendere il layout della webapp non fisso, in modo che l'applicazione occupi tutta l'area visibile del browser web. In questo modo utenti dotati di monitor ad alta risoluzione possono sfruttare a pieno tutta l'area dello schermo.

Le opzioni per la selezione della lingua sono state posizionate nella parte superiore destra di ogni pagina, in modo tale che siano ben visibili e sempre accessibili.

Sempre nella parte superiore destra è stato posizionato il blocco di login/logout e come per la selezione della lingua anche questo blocco gode dei vantaggi offerti da tale posizionamento.

Il risultato è visibile in Figura 5.3, dove vengono messi in risalto gli elementi minimi di composizione del layout. L'immagine evidenzia, inoltre, alcuni piccoli accorgimenti che sono stati adottati in fase di sviluppo, come l'utilizzo del colore azzurro per i link (gli utenti sono infatti abituati ad identificare i link con questo colore), o la selezione di un colore differente per la voce di menu attiva.

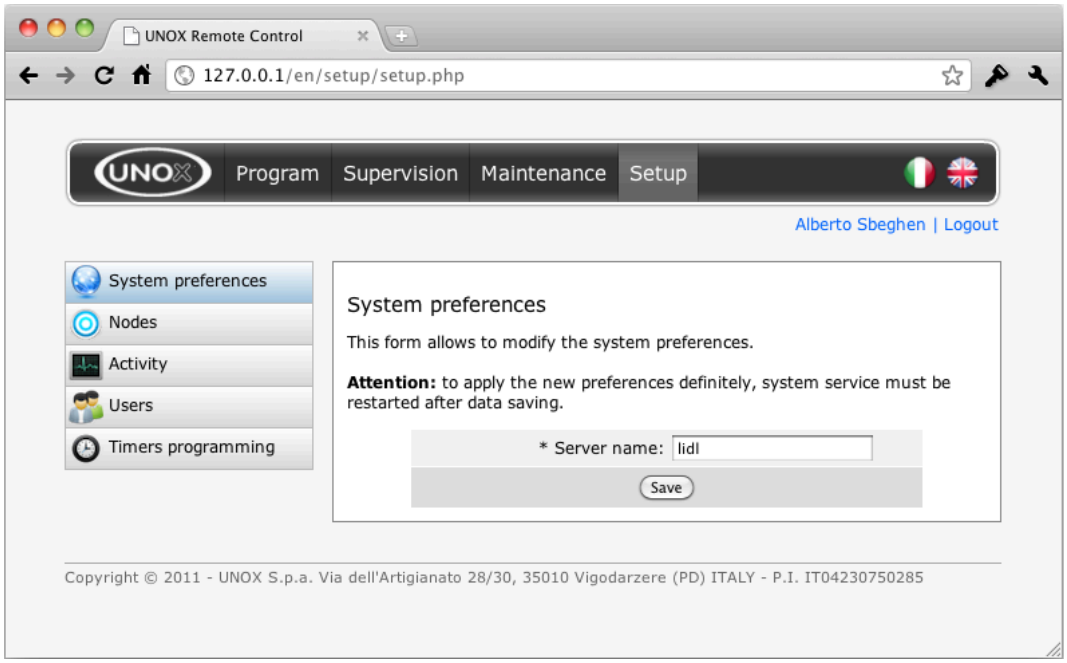

Figura 5.4: Struttura dei contenuti dell'applicativo di interfaccia utente con tab verticali.

I contenuti propri di ogni sezione sono stati organizzati in una struttura con dei tab verticali. L'effetto risultante è riportato in Figura 5.4, la quale evidenzia anche il risultato della visualizzazione dei contenuti in lingua inglese. In sostanza ogni funzionalità interna ad una determinata sezione può essere raggiunta selezionando il relativo tab laterale, il quale fa sì che il riquadro con i relativi contenuti compaia nell'area centrale-destra della pagina. Anche in questo caso il posizionamento dei pulsanti relativi ai diversi tab non è casuale, in quanto sfrutta l'area calda di sinistra, che dopo la zona superiore risulta essere la parte della pagina più vista dall'utente.

5.4.2 Gestione delle richieste AJAX e i loading box

Come è stato detto in precedenza, l'interfaccia utente di URC fa larghissimo uso di AJAX per aggiornare dinamicamente gli elementi contenuti nelle diverse pagine. AJAX, quando deve recuperare nuovi dati, evade una richiesta al server in background per ottenere l'XML (o la stringa) di risposta. Al ricevimento dell'XML le librerie AJAX, se necessario, avvieranno automaticamente la funzione JavaScript incaricata di modificare la pagina (ad esempio facendo uso di jQuery) per visualizzare i nuovi dati richiesti. Considerando però che la richiesta HTTP effettuata da AJAX viene eseguita in background di default l'utente non ha un riscontro diretto sull'effettiva attività in corso.

Ad esempio, se l'attività viene avviata alla pressione di un link, nell'arco di tempo successivo al click (fino al ricevimento dei dati da parte del server) l'utente non può capire se la richiesta AJAX è stata effettivamente ricevuta dal server e se

questo sta elaborando la richiesta. Il rischio che si corre è che l'utente, stanco di aspettare, abbandoni la pagina, o effettui click multipli sul link per ottenere quello che desidera. Tutto questo, se non valutato attentamente, rischia di appesantire l'esperienza di utilizzo dell'utente.

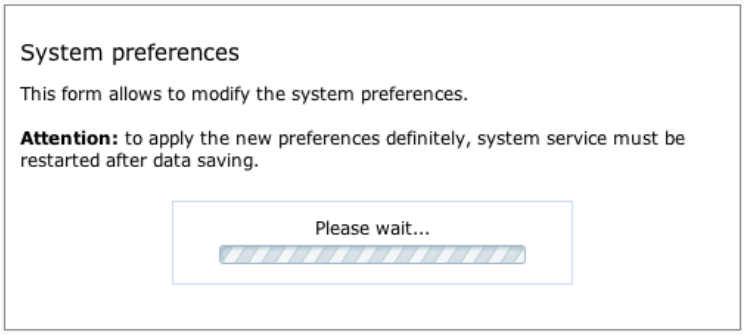

Figura 5.5: Esempio di un riquadro di loading per la corretta gestione delle richieste AJAX.

Per evitare il problema in URC l'avvio di ogni richiesta AJAX determina la comparsa sullo schermo di un riquadro che segnala all'utente che l'attività è stata avviata e che i dati stanno per essere elaborati. Il riquadro visualizzato presenta sempre una GIF animata che dà l'idea che l'attività è in fase di elaborazione. Un esempio del risultato è presentato in Figura 5.5, la quale evidenzia il caricamento del modulo con le preferenze di sistema (visualizzato in Figura 5.4).

5.5 Struttura dell'applicativo di interfaccia

Passiamo ora all'analisi di alcune parti del codice dell'applicativo web-based, focalizzando l'attenzione su determinate parti del progetto.

Finora non è stato esplicitamente detto, ma banalizzando si può affermare che l'applicativo di interfaccia non è altro che una raccolta di maschere diverse costituite da una serie di form HTML. In ambito web con il temrine form si definiscono tutti i moduli di inserimento e modifica dei dati. Solitamente un form è composto da una serie di caselle di testo, checkbox, combobox di selezione, pulsanti, ecc.

Questo aspetto, seppur rappresenti un'astrazione di alto livello, può essere sfruttato per mettere a punto un motore di gestione dei contenuti generalizzato, in modo tale che non si debbano scrivere migliaia di righe di codice che in realtà svolgono lo stesso lavoro.

Per tale ragione durante le fasi di sviluppo della webapp si è deciso di strutturare i contenuti in modo tale da seguire delle linee guida comuni, che, se osservate, possono portare alla costruzione di un unico motore di gestione dei form. In sostanza bisogna cercare di strutturare i form tutti allo stesso modo, standardizzando inoltre le opzioni disponibili.

Come vedremo, questo ha portato alla costituzione di un'architettura per l'applicazione nella quale vi è una netta divisione fra logica di interfaccia e motore PHP

per la gestione dei dati. Questa netta divisione offre diversi vantaggi, come una più semplice interpretazione del codice, un'interfaccia standardizzata e lo spreco di minori risorse per l'aggiunta in futuro di nuovi moduli.

5.5.1 Il database MySQL

Nella sezione 4.3.1 si è visto come è stato progettato il database MySQL per la parte riguardante il servizio di sistema e il suo funzinamento in accoppiata alla webapp di interfaccia. Ma proprio l'applicativo di interfaccia, però, ha bisogno di memorizzare una serie di dati utili solo ai fini del suo funzionamento, i quali vengono salvati in delle tabelle dedicate.

I dati che servono di supporto all'applicativo di interfaccia utente sono i seguenti:

- dati degli utenti abilitati ad effettuate l'accesso al sistema (procedura di login);
- *ricette* per la gestione dei programmi di cottura dei forni.

Mentre il primo punto risulta chiaro, il secondo non è ancora stato presentato.

Supponendo che l'utilizzatore sia interessato a gestire i programmi di cottura dei forni creando una serie di "insiemi di programmi di cottura" standard, risulta opportuno mettere a disposizione dell'utente un apposito modulo di creazione delle ricette. In sostanza attraverso tale modulo l'utente può creare i propri programmi di cottura standard ed includerli in dei gruppi predefiniti (le ricette, appunto), in modo tale da poter programmare un determinato insieme di forni selezionando la ricetta di interesse. La ricetta può essere quindi paragonata ad un'immagine della memoria interna ai forni, nella quale ogni programma di cottura occupa una posizione ben definita. Ad esempio la ricetta denominata "NordEst-Italia" può contenere alla posizione 1 il programma di cottura del pollo, alla posizione 2 il programma di cottura delle patate e così via.

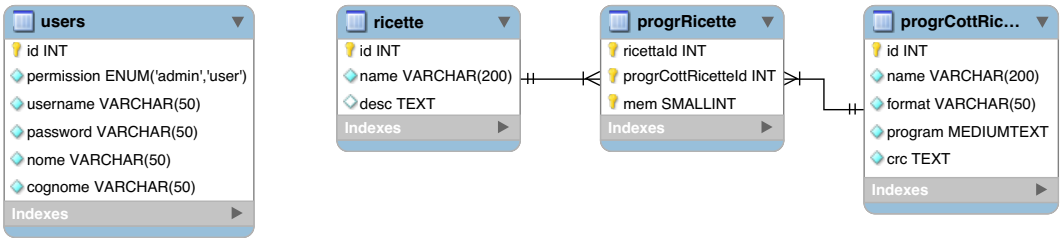

Figura 5.6: Tabelle del database MySQL di supporto all'applicazione di interfaccia utente.

Considerando che la progettazione delle tabelle del database risulta alquanto semplice, in Figura 5.6 si riporta direttamente il diagramma tabellare MySQL.

5.5.1.1 Users

La tabella *users* è dedicata al salvataggio degli utenti abilitati ad accedere al pannello web-based. La struttura della tabella è alquanto semplice e necessita di una

precisazione solo per evidenziare che per il momento sono stati previsti due gradi di privilegi utente: utente standard (*user*) e utente amministratore (*admin*). In pratica si è deciso che l'utente standard non ha accesso all'area *Setup* della webapp. In futuro, se ve ne fosse la necessità, si può modificare la tabella ed introdurre privilegi dedicati ad ogni sezione dell'interfaccia, in modo tale che ogni utente possa accedere solo alle funzionalità che lo competono.

Il codice SQL di creazione della tabella è il seguente:

```
1  CREATE TABLE IF NOT EXISTS `users`
2  (
3    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
4    `permission` ENUM('admin','user') NOT NULL ,
5    `username` VARCHAR(50) BINARY NOT NULL ,
6    `password` VARCHAR(50) NOT NULL ,
7    `nome` VARCHAR(50) NOT NULL ,
8    `cognome` VARCHAR(50) NOT NULL ,
9    PRIMARY KEY (`id`) ,
10   UNIQUE INDEX `username_UNIQUE` (`username` ASC)
11 ) ENGINE = InnoDB;
```

5.5.1.2 Ricette, ProgrRicette e ProgrCottRicette

Questa tripletta di tabelle consente di implementare l'archiviazione delle ricette così come è stata presentata in precedenza.

La tabella *progrCottRicette*, che risulta molto simile alla tabella *progrCott* presentata alla sezione 4.3.1, permette di salvare i programmi di cottura standard. La tabella *ricette* permette invece di salvare l'elenco delle ricette utilizzate dai diversi punti vendita, ognuna delle quali viene contraddistinta da un nome (e da una descrizione). Ciò che lega le ricette con i programmi di cottura è la tabella *progrRicette* (diminutivo di *programmazione delle ricette*), la quale associa ogni ricetta a diversi programmi di cottura. Ogni associazione viene contraddistinta dall'attributo *mem*, che identifica la posizione di memorizzazione del programma di cottura all'interno del forno.

I codici SQL di creazione delle tabelle sono riportati di seguito:

```
1  CREATE TABLE IF NOT EXISTS `ricette`
2  (
3    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
4    `name` VARCHAR(200) NOT NULL ,
5    `desc` TEXT NULL ,
6    PRIMARY KEY (`id`) ,
7    UNIQUE INDEX `name_UNIQUE` (`name` ASC)
8  ) ENGINE = InnoDB;
9
10 CREATE TABLE IF NOT EXISTS `progrCottRicette`
11 (
12   `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
13   `name` VARCHAR(200) BINARY NOT NULL ,
14   `format` VARCHAR(50) NOT NULL ,
```

```
15    `program` MEDIUMTEXT NOT NULL ,
16    `crc` TEXT NOT NULL ,
17    PRIMARY KEY (`id`) ,
18    UNIQUE INDEX `name_UNIQUE` (`name` ASC)
19  ) ENGINE = InnoDB;
20
21  CREATE TABLE IF NOT EXISTS `progrRicette`
22  (
23    `ricettaId` INT UNSIGNED NOT NULL ,
24    `progrCottRicetteId` INT UNSIGNED NOT NULL ,
25    `mem` SMALLINT UNSIGNED NOT NULL ,
26    PRIMARY KEY (`ricettaId`, `progrCottRicetteId`, `mem`) ,
27    INDEX `progrRicetteRicettaId` (`ricettaId` ASC) ,
28    INDEX `progrRicetteProgrCottRicetteId` (`progrCottRicetteId` ASC) ,
29    CONSTRAINT `progrRicetteRicettaId`
30      FOREIGN KEY (`ricettaId`)
31      REFERENCES `ricette` (`id`)
32      ON DELETE CASCADE
33      ON UPDATE NO ACTION,
34    CONSTRAINT `progrRicetteProgrCottRicetteId`
35      FOREIGN KEY (`progrCottRicetteId`)
36      REFERENCES `progrCottRicette` (`id`)
37      ON DELETE CASCADE
38      ON UPDATE NO ACTION
39  ) ENGINE = InnoDB;
```

5.5.2 Organizzazione del codice PHP

Il motore di scripting PHP si divide in molti file. Nel progetto troviamo file fondamentalmente semplici dedicati alla "stampa" dell'interfaccia grafica e file più complessi,dedicati al prelievo e al salvataggio dei dati sul database MySQL. Mentre la seconda famiglia di script verrà presentata alla prossima sezione del testo, in questa parte si vuole illustrare l'organizzazione di base del codice che si occupa di gestire tre funzionalità principali del programma:

- visualizzazione dell'interfaccia e del layout della webapp;
- gestione dei contenuti in più lingue (completando l'analisi svolta alla sezione 5.2.2);
- gestione dell'autenticazione degli utenti.

5.5.2.1 Visualizzazione dell'interfaccia

Come si è visto in precedenza, il layout della webapp risulta alquanto semplice, poiché risulta composto fondamentalmente da tre blocchi distinti: header, content e footer.

Come è facile supporre, l'applicazione web-based è composta da diverse pagine, ognuna delle quali presenta gli stessi blocchi header e footer. Pertanto risulta assai conveniente fare in modo che il codice di generazione dell'HTML per l'impaginazione di questi due blocchi sia comune a tutte le pagine, in modo tale che eventuali modifiche future non comportino una revisione del codice contenuto in ogni pagina del progetto.

A tale scopo si è pensato di racchiudere il codice HTML di questi due blocchi all'interno di due file PHP, i quali verranno inclusi nel file che costituisce ogni pagina.

Tramite la funzione *include();* del PHP è infatti possibile includere all'interno di uno script PHP l'intero contenuto di un altro file, come avviene ad esempio con la direttiva *#include* del linguaggio C.

A titolo di esempio si riporta il codice PHP+HTML di una qualsiasi pagina del progetto (priva di contenuto):

```
1  <?php
2  define("ROOT_URL", ".");
3  include(ROOT_URL.'/init.php');
4  ?>
5  <!DOCTYPE html PUBLIC "-//W3C//DTD␣XHTML␣1.0␣Transitional//EN" "http://
       www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
6  <html>
7  <head>
8    <title>Pagina d'esempio</title>
9  ␣␣<link␣rel="stylesheet"␣type="text/css"␣media="all"␣href="/style.css"␣
       />
10 </head>
11 <body>
12 <?php
13 include(ROOT_URL.'/header.php');
14 ?>
15 <h2>Pagina␣d'esempio</h2>
16 Contenuto...
17 <?php
18 include(ROOT_URL.'/footer.php');
19 ?>
20 </body>
21 </html>
```

Alle righe 9 e 14 del codice si trovano le due direttive include, per aggiungere alla pagine rispettivamente il blocco header (file *header.php*) e il blocco footer (file *footer.php*).

5.5.2.2 Gestione multilingua

Alla sezione 5.2.2 si è visto come il *ModRewrite* di Apache possa essere applicato alla gestione dei contenuti multilingua per trasportare in modo implicito il parametro GET denominato *lang*.

Ora vedremo come il motore di scripting PHP di URC raccolta tale informazione per far funzionare l'intero sistema di localizzazione.

Prima di tutto va detto che la gestione multilingua comporta che tutti i contenuti testuali delle diverse pagine devono poter essere recuperati da una qualche "base di dati" e presentati runtime all'occorrenza. Questo implica fondamentalmente tre cose:

- le pagine non devono contenere stringhe di testo statiche che non possono essere tradotte;

- le pagine non possono contenere immagini con testi, a meno che queste non vengano salvate nelle diverse lingue;
- il layout di ogni pagina si deve poter adattare alla lunghezza del testo, che nelle diverse lingue può variare sensibilmente.

Per queste ragioni la webapp di URC non contiene testi statici né immagini contenenti testi; inoltre il layout di ogni pagina è stato impaginato tramite i fogli di stile CSS in modo tale che si possa adattare completamente al contenuto.

Per quanto riguarda l'archiviazione dei testi nelle varie lingue si possono scegliere due soluzioni differenti:

- salvare i testi in una apposita tabella del database MySQL; ogni testo dovrà essere identificato da un'apposita chiave composta dalla coppia *(nome_stringa, lingua)*;
- inserire i testi in un array PHP che viene caricato il memoria primaria del server durante l'interpretazione dello script PHP.

La prima soluzione, che a prima vista può sembrare la migliore, in realtà nasconde diverse problematiche, prima fra tutte la scarsa efficienza. Si pensi infatti di avere una pagina con 20 testi differenti; dovendoli prelevare dal database all'occorrenza si rischia che il codice PHP debba effettuare ben 20 query SQL prima di riuscire a comporre l'intera pagina. Pertanto è facile comprendere che è impensabile applicare la prima soluzione al caso in esame, a meno che non si decida di creare una mappatura del contenuto della tabella MySQL in una forma interpretabile da PHP con un minor dispendio di tempo e risorse, come ad esempio un array.

Ecco dunque che le due soluzioni coincidono, in quanto la prima presuppone comunque l'utilizzo della seconda. Chiaramente si può anche ipotizzare di implementare la prima soluzione senza far uso di array, ma questo può aver senso per pagine in cui il numero di diversi testi è limitato e ogni testo presenta una lunghezza minima di 300-400 caratteri[2].

Considerando però che in URC il numero di testi che compone la webapp è ridotto si è deciso di archiviare tutti i contenuti direttamente su una struttura array multidimensionale. L'array in questione è incluso nel file *dictionary.php* e di seguito se ne riporta un breve estratto:

```
1 <?php
2 $italiano = "ita";
3 $english = "en";
4
5 //************************ ITALIANO ************************//
6 $dictionary[$italiano]["sysName"] = "UNOX Remote Control";
7 $dictionary[$italiano]["generalError"] = "Si &egrave; verificato un
    errore generico. Per favore provare a ripetere l\'operazione...";
8 $dictionary[$italiano]["salva"] = "Salva";
```

[2]questa soluzione viene adottata in tutti i più diffusi CMS di gestioene dei blog, dove il testo presente in ogni pagina viene racchiuso in un'unica riga all'interno della base di dati.

```
9  $dictionary[$italiano]["comandi"] = "Comandi";
10 $dictionary[$italiano]["noDbRow"] = "Nessun␣record␣presente␣in␣archivio
       ";
11 //************************ ENGLISH ************************//
12 $dictionary[$english]["sysName"] = "UNOX␣Remote␣Control";
13 $dictionary[$english]["generalError"] = "A␣generic␣error␣accours.␣
       Please␣try␣again...";
14 $dictionary[$english]["salva"] = "Save";
15 $dictionary[$english]["comandi"] = "Actions";
16 $dictionary[$english]["noDbRow"] = "No␣row␣in␣the␣archive";
17 ?>
```

La logica di gestione dei contenuti nelle diverse lingue è invece delegata al seguente blocco di codice PHP, inserito nel file *init.php* caricato all'inizio di ogni pagina:

```
1  <?php
2  //*** Language
3  $lang = "ita";
4  if (isset($_GET["lang"]))
5    $lang = $_GET["lang"];
6
7  //*** Caricamento dizionario traduzioni
8  include(ROOT_URL.'/dictionary.php');
9  function dict($key)
10 {
11   global $dictionary, $lang, $italiano;
12   if (isset($dictionary[$lang][$key]))
13     return $dictionary[$lang][$key];
14   return $dictionary[$italiano][$key];
15 }
16 ?>
```

In sostanza nelle prime righe di codice viene raccolto il valore contenuto nel parametro *lang* della stringa GET; se il parametro non è specificato la lingua viene automaticamente impostata in italiano. Il parametro *lang*, come è stato visto alle sezioni precedenti, viene generato implicitamente dal *ModRewrite* di Apache.

Nel seguito del codice il PHP carica l'intero array *$dictionary* con il dizionario dei testi tradotti e attraverso la funzione *dict* restituisce la stringa con chiave *$key* nella lingua impostata nella variabile *$lang*.

Quindi, concludendo, se si desidera stampare la stringa relativa all'errore generico utilizzando la lingua impostata, sarà sufficiente richiamare scrivere il seguente codice PHP:

```
1  <?php
2  define("ROOT_URL", ".");
3  include(ROOT_URL.'/init.php');
4
5  echo "<h3>".dict("generalError")."</h3>";
6  ?>
```

5.5.2.3 Autenticazione degli utenti

L'autenticazione degli utenti basa il proprio funzionamento sull'utilizzo delle sessioni PHP. La sessione si può definire come l'arco di tempo in cui viene monitorata la connessione di un utente. Durante questo arco di tempo è possibile conservare informazioni sulla navigazione accessibili da ogni pagina collegata alla sessione. Più semplicemente la sessione inizia quando l'utente accede all'applicazione web-based e finisce quando la abbandona, chiudendo il browser. Le sessioni PHP rappresentano la logica che ad esempio sta dietro le procedure di login nei portali e nei forum, dietro i carrelli della spesa (virtuali) e dietro i contatori di visitatori online.

Una volta che una sessione viene aperta, il PHP vi attribuisce un ID (chiave numerica) univoco, in modo tale che durante la navigazione il browser lo possa trasmettere ad ogni nuova richiesta HTTP al server Apache per l'identificazione della sessione (e quindi dell'utente). In PHP la sessione può essere avviata invocando la funzione *session_start();*. In URC l'invocazione di tale funzione è stata inclusa nel file *init.php* che come è stato già detto viene incluso all'inizio di ogni pagina dell'applicativo di interfaccia.

All'interno della sessione si possono salvare una serie di informazioni utilizzando l'array del PHP *$_SESSION[]*, informazioni che possono essere recuperate durante tutta la durata della sessione.

La webapp di URC sfrutta proprio questo principio per implementare la procedura di autenticazione degli utenti e lo fa nel modo seguente:

- l'utente che accede alla pagina *login.php* inserisce i propri username e password;
- la pagina tramite AJAX invia i dati alla pagina *loginExecute.php*, la quale effettua la connessione alla tabella *users* del database MySQL per verificare se i dati di login inseriti sono corretti;
- se il login ha esito positivo la pagina *loginExecute.php* salva le seguenti variabili di sessione:

```
1 $_SESSION["login-user"] = $_POST["username"];
2 $_SESSION["login-id"] = $id;
3 $_SESSION["login-permission"] = $permission;
4 $_SESSION["login-name"] = $nome."␣".$cognome;
```

Le variabili PHP *$id*, *$permission*, *$nome* e *$cognome* vengono recuperate dai valori della tabella *users*, mentre la variabile *$_POST["username"]* corrisponde all'username inserito in fase di autenticazione (trasmesso via POST da AJAX).

Pertanto, se si vuole fare in modo che una determinata pagina possa essere visitata solo dopo che l'utente ha eseguito la procedura di login, (cosa che in URC avviene per tutte le pagine dell'applicazione ad esclusione di *login.php*) è sufficiente includere nelle prime righe di codice della pagina le seguenti direttive:

```
1 <?php
2 define("ROOT_URL", "..");
3 include(ROOT_URL.'/init.php');
4 include(ROOT_URL.'/loginCheck.php');
5 ?>
```

La pagina *loginCheck.php* riportata di seguito verifica se la variabile *$_SESSION['login-user']* è correttamente inizializzata. Se così non è il PHP provvede ad effettuare un redirect alla pagina di autenticazione:

```
1 <?php
2 if (!isset($_SESSION['login-user']))
3 {
4   header("Location: ".ROOT_URL."/login.php");
5   exit;
6 }
7 ?>
```

5.5.3 Le librerie per la gestione dei form HTML

Come è stato detto in precedenza, la webapp di URC contiene una serie di pagine, ognuna delle quali (ad esclusione di qualche raro caso) include un form HTML per l'inserimento e la modifica dei dati. La gestione delle preferenze di sistema, la configurazione dei nodi diretti, la gestione dei programmi di cottura, la gestione dei parametri di funzionamento sono tutti esempi di attività che richiedono l'utilizzo di un form HTML.

Per tale ragione in URC si è pensato di scrivere una serie di routine JavaScript ad alto livello che attraverso AJAX permettano di eseguire tutte le operazioni legate alla modifica dei dati tramite form. In questo modo ci sarà una sola logica di funzionamento comune a tutta l'applicazione, sia lato utente (interfaccia standardizzata) sia lato programmazione.

Vediamo dunque quali sono gli elementi che costituiscono una maschera di modifica dei dati e come questi sono organizzati in un classico form della webapp di URC (vedi Figura 5.7):

- Tabella con i dati presenti nel database di sistema: la tabella serve per visualizzare all'utente i dati già salvati e per consentirgli di selezionare le righe che intende modificare o eliminare. Premendo il comando di eliminazione il motore JavaScript (con AJAX) invia una richiesta al server ed cancella dinamicamente la riga dalla tabella, senza alcun refresh della pagina.

- Form HMTL per la modifica dei dati: il form riporta una serie di campi testuali o di selezione (o di altre tipologie), i quali generalmente rispecchiano gli attributi delle tabelle del database. Il form viene utilizzato sia per modificare i dati esistenti sia per inserire nuove entry. Nel primo caso, cliccando sul comando di modifica in tabella, il motore JavaScript provvede ad inviare una richiesta AJAX al server per il caricamento dei dati relativi a quella riga. Una volta che i dati sono stati ricevuti vengono caricati all'interno dei campi del form.

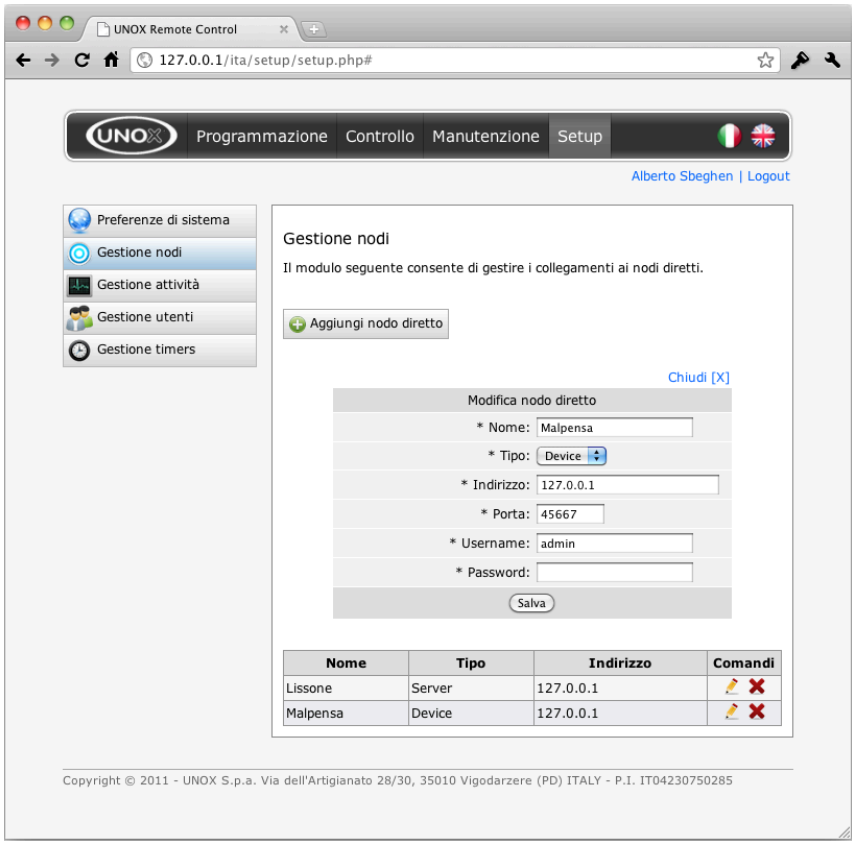

Figura 5.7: Classico form HTML dell'applicativo di interfaccia utente.

- Pulsante di aggiunta di una nuova entry: il pulsante consente di effettuare un reset del form HTML per l'inserimento di una nuova entry.

Va evidenziato che al caricamento della pagina il form HTML viene mantenuto nascosto; viene invece presentato con un piccolo effetto dissolvenza solo quando si preme sul comando di modifica della tabella o quando si preme il pulsante per l'aggiunta di una nuova entry. Un discorso simile vale per la tabella di presentazione dei dati del database, la quale in seguito al caricamento della pagina compare in sostituzione ad un box di loading solo quando i dati sono stati effettivamente prelevati con successo.

Passiamo ora ad analizzare la struttura delle diverse funzioni JavaScript preposte al caricamento e alla modifica dei dati. Per ogni funzione verrà riportata una semplice descrizione che ne riassume le funzionalità e un listato in pseudo-codice che ne illustra il funzionamento. Prima di elencare le diverse routine JavaScript, però, analizziamo il codice JavaScript dedicato alla gestione di una richiesta AJAX con XML.

5.5.3.1 Richiesta AJAX

Il codice JavaScript necessario a realizzare una richiesta AJAX con XML è il seguente[3]:

```
1  var xmlHttpReq = false;
2  var self = this;
3  // Mozilla/Safari
4  if (window.XMLHttpRequest)
5    self.xmlHttpReq = new XMLHttpRequest();
6  // IE
7  else if (window.ActiveXObject)
8    self.xmlHttpReq = new ActiveXObject("Microsoft.XMLHTTP");
9  self.xmlHttpReq.open('POST', 'send.php', true);
10 self.xmlHttpReq.setRequestHeader('Content-Type', 'application/x-www-
       form-urlencoded');
11 self.xmlHttpReq.onreadystatechange = function()
12 {
13   if (self.xmlHttpReq.readyState == 4)
14   {
15     if (self.xmlHttpReq.status == 200)
16     {
17       var xmlResp = self.xmlHttpReq.responseXML.documentElement;
18       // L'oggetto XML con la risposta del server sta in xmlResp
19     }
20   }
21 }
22 self.xmlHttpReq.send('par1=1&par2=2');
```

Nelle prime righe di codice, in base al browser utilizzato, viene creato l'oggetto JavaScript per la gestione delle richieste AJAX.

Alla riga 9 vengono invece definiti il metodo utilizzato (POST o GET) e la pagina a cui inviare la richiesta (nell'esempio *send.php*).

Tramite la funzione *setRequestHeader* si possono invece definire degli header specifici da inviare con la richiesta HTTP di AJAX.

Per inviare la richiesta AJAX occorre invocare il metodo *send* al quale, solo nel caso di comunicazione POST, si può passare la lista delle variabili POST da trasmettere tramite HTTP (nell'esempio *par1* e *par2*).

Con il parametro *onreadystatechange* si può invece definire una funzione JavaScript di callback che viene automaticamente richiamata da AJAX quando lo stato della connessione (che può essere letto dalla variabile *readyState*) subisce una variazione. Gli stati possibili sono 5:

- 0 - Uninitialized: l'oggetto XMLHttpRequest esiste, ma non è stato richiamato alcun metodo per inizializzare una comunicazione;
- 1 - Open: è stato richiamato il metodo open() ed il metodo send() non ha ancora effettuato l'invio dati;

[3]si evidenzia che le risposte alle richieste AJAX possono essere gestite anche come semplici stringhe, per questo si specifica che in URC AJAX viene utilizzato in accoppiata a XML.

- 2 - Sent: il metodo send() è stato eseguito ed ha effettuato la richiesta;
- 3 - Receiving: i dati in risposta cominciano ad essere letti;
- 4 - Loaded: l'operazione è stata completata.

Pertanto, monitorando lo stato 4 e controllando il codice relativo allo status HTTP della richiesta (leggibile attraverso la variabile *status*) si può reperire l'XML con la risposta dalla variabile *responseXML*.

5.5.3.2 FillTable

La funzione *FillTable*, la prima delle routine JavaScript atte alla gestione dei form HTML in URC, consente di caricare i dati ricevuti tramite AJAX da un apposito script PHP in una specifica tabella HTML. Il codice, come quello delle altre funzioni, basa il proprio funzionamento sulla richiesta AJAX appena vista. Ma ciò che differenzia la funzione dalle altre sta proprio nella routine di caricamento dei dati dopo che questi sono stati ricevuti dal server tramite l'esecuzione un apposito script PHP.

Le linee guida del codice della funzione sono le seguenti:

```
1  function FillTable (in nomeTabella, in paginaPrelievoDati)
2  {
3    Invia la richiesta AJAX alla pagina PHP paginaPrelievoDati
4    Nascondi la tabella nomeTabella
5    Fai comparire il loading box
6
7    Quando ricevi i risultati (ajax.readyState = 4)
8    {
9      Nascondi il loading box
10     if (risultato = Ok)
11     {
12       Rimuovi tutte le righe della tabella (tranne quelle di
             intestazione)
13       if (numero elementi = 0)
14         Aggiungi alla tabella una sola riga che indichi che non ci sono
               elementi
15       else
16       {
17         Per ogni elemento ricevuto via XML
18         {
19           Inserisci il tag <tr> di nuova riga
20           Per ogni tag dell'elemento
21             Inserisci il tag <td></td> di nuova colonna con all'interno
                   il valore del tag dell'elemento
22           Inserisci il tag </tr> di chiusura nuova riga
23         }
24       }
25       Visualizza la tabella nomeTabella
26     }
27     else
28       Visualizza box di errore
29   }
30 }
```

Perché tutto funzioni correttamente occorre che il numero delle colonne della tabella coincida con il numero di tag interni ad ogni elemento passato via XML ad AJAX. Ciò significa, ad esempio, che in presenza della tabella con le colonne Username, Nome e Cognome la pagina PHP di generazione dell'XML contenente i dati prelevati dal database dovrà generare un listato di questo tipo:

```
<elenco>
  <elemento>
    <username>user1</username>
    <nome>nome1</nome>
    <cognome>cognome1</cognome>
  </elemento>
  [...]
  <elemento>
    <username>userN</username>
    <nome>nomeN</nome>
    <cognome>cognomeN</cognome>
  </elemento>
</elenco>
```

In questo modo il contenuto dei tre tag di ogni elemento verrà copiato secondo l'ordine nelle relative colonne della tabella.

Va evidenziato che il prelievo del tag TABLE dal codice HTML e di ogni altro tag necessario avviene utilizzando le librerie jQuery.

5.5.3.3 FillForm

La funzione *FillForm* funziona in modo simile alla funzione *FillTable* e si differenzia da quest'ultima solo per la funzione definita in *onreadystatechange* di AJAX.

La funzione *FillTable*, come è stato visto, ignora completamente il nome dei tag del listato XML, poiché si limita a copiarli nelle colonne della tabella in base all'ordine con cui questi vengono trasmessi.

FillForm, invece, una volta rcevuto l'XML preleva il tag *input* del relativo form che presenta l'attributo *name* uguale al nome del tag ricevuto (questa operazione avviene tramite jQuery). Pertanto il file PHP di generazione dell'XML dovrà utilizzare per i tag lo stesso nome che compare negli attributi *name* dei diversi *input* del form HTML.

Il codice che realizza quanto appena detto è il seguente:

```
function FillForm (in nomeForm, in paginaPrelievoDati)
{
  Invia la richiesta AJAX alla pagina PHP paginaPrelievoDati
  Nascondi il form HTML nomeForm
  Fai comparire il loading box

  Quando ricevi i risultati (ajax.readyState = 4)
  {
    Nascondi il loading box
    if (risultato = Ok)
```

```
11      {
12        Reset del form HTML
13        Per ogni elemento ricevuto via XML
14        {
15          nomeTag = nome dell'elemento ricevuto
16          valoreTag = valore dell'elemento ricevuto
17          tagInput = Preleva tramite jQuery il tag input con nome nomeTag
18          Imposta l'attributo value di tagInput a valoreTag
19        }
20        Visualizza il form HTML nomeForm
21      }
22      else
23        Visualizza box di errore
24    }
25  }
```

5.5.3.4 SendForm

Questa funzione viene usata ogni qualvolta risulta necessario inviare i contenuti di un form ad una pagina PHP, la quale, ad esempio, provvederà a salvare i dati nel database di sistema.

La funzione solitamente viene invocata alla pressione del pulsante di invio (submit) di un form HTML.

La struttura è identica a quella usata per l'invio di una richiesta AJAX e il cuore sta tutto nell riga di codice JavaScript relativa alla trasmissione dei parametri via POST:

```
1  self.xmlHttpReq.send($('#'+formName).serialize());
```

Il metodo *serialize()* di jQuery, come riportato nella documentazione ufficiale, esegue la seguente funzione: "Encode a set of form elements as a string for submission". In sostanza attraverso la funzione *$('#'+formName)* si preleva l'oggetto form con attributo *id* uguale a *formName* dal codice HTML della pagina. Attraverso il metodo *serialize*, invece, tutte le coppie *(nome-campo, valore)* del form vengono mappate in una stringa di trasmissione POST.

Il codice PHP che riceve tali informazioni otterrà dunque un array chiave-valore con i valori inseriti in tutti i campi del form HTML.

Capitolo 6

Il bridge Ethernet

Dopo aver progettato il servizio di sistema, software preposto allo scambio di informazioni fra i diversi nodi gerarchici, e dopo aver studiato l'implementazione della webapp di interfaccia utente, viene analizzato il *bridge Ethernet* di controllo.

Come è stato anticipato nei capitoli introduttivi, il bridge Ethernet è la scheda elettronica che viene installata a bordo del forno e che permette al sistema URC di interfacciarsi alla sua logica di controllo. La scheda in questione è stata denominata bridge Ethernet per enfatizzare il fatto che le comunicazioni per la ricetrasmissione dei dati avvengono, appunto, tramite interfaccia Ethernet.

Il bridge è una scheda elettronica provvista di un microcontrollore e di una serie di interfacce; per cui in realtà è l'unico elemento del sistema a disporre sia di una parte hardware che software.

Va precisato fin da subito che sebbene la scheda e parte del software vengano presentati in questo elaborato, lo sviluppo di questa parte del sistema ha interessato limitatamente l'attività di tirocinio in UNOX per tre ragioni principali:

- la progettazione dell'hardware è stata svolta interamente da UNOX poiché non era prevista nelle attività di tirocinio;

- lo sviluppo dell'intero Stack TCP/IP, ovvero del modulo software dedicato alla gestione dell'omonima pila protocollare, è stato sviluppato da un gruppo di lavoro del CNR (Centro Nazionale delle Ricerche) coordinato dal Prof. Ivan Cibrario Bertolotti; come per la progettazione dell'hardware, anche questa parte non rientra nelle attività previste dal lavoro di tesi;

- le attività di sviluppo della scheda elettronica definitiva (totalmente funzionante) e di collaudo di una versione basilare del relativo firmware di controllo (con Stack TCP/IP) sono state portate a termine con successo solo al termine dell'esperienza di tirocinio.

Quanto riportato all'ultimo punto non ha dunque permesso di terminare lo sviluppo dell'intero firmware di controllo, e per eseguire le fasi di test si sono svolte solo delle simulazioni.

In ogni caso, come verrà discusso in seguito, benché si tratti di simulazioni si sono potute analizzare delle caratteristiche del sistema che ne evidenziano, oltre che il corretto funzionamento, le elevate performance.

6.1 L'hardware di sistema

L'hardware del bridge Ethernet è costituito da una singola scheda elettronica multistrato, la quale risulta composta da diversi blocchi logici. Lo schema che evidenzia i principali moduli che compongono la scheda è riportato in Figura 6.1. Ogni blocco identifica uno o più circuiti integrati, provvisti del relativo hardware di interfaccia. Di seguito viene riportata una breve descrizione del contenuto di ogni blocco.

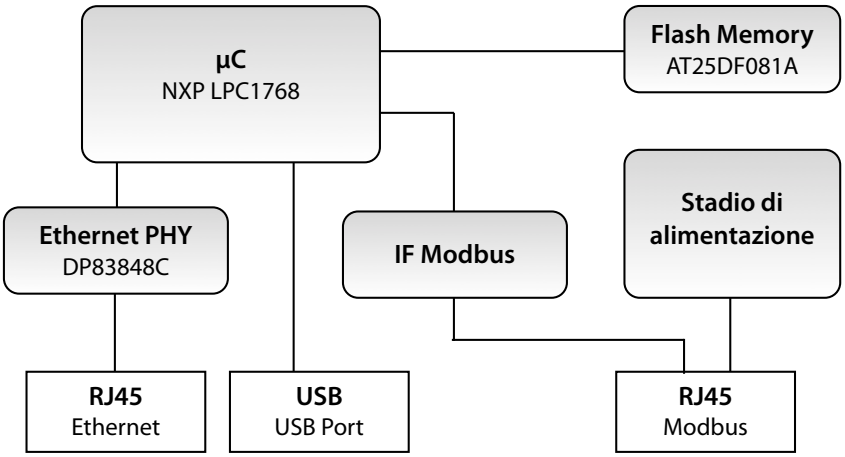

Figura 6.1: Schema a blocchi del bridge Ethernet.

Stadio di alimentazione Lo stadio di alimentazione è un blocco che solitamente compare in ogni scheda elettronica. Ha la funzione di adattare e stabilizzare la tensione di alimentazione al livello richiesto dall'elettronica di controllo. Solitamente in questo blocco vengono inclusi anche dei dispositivi di protezione, quali ad esempio diodi o fusibili.

Nel caso del bridge di UNOX lo stadio di alimentazione è di tipo lineare e presenta più di un integrato per la stabilizzazione della tensione. La tensione primaria a +12V viene prelevata direttamente dal bus Modbus.

Microcontrollore Il microcontrollore è un LPC1768 della NXP. Presenta un'architettura ARM Cortex-M3 a 32 bit e 64 KByte di memoria primaria SRAM.

Questo integrato è stato scelto per la varietà di interfacce e moduli di cui dispone: 512 KByte di memoria Flash interna per il firmware, un controller Ethernet 10/100, un controller USB 2.0, svariate porte CAN, I2S, ADC a 12 bit e DAC a 10 bit; la frequenza di clock può arrivare fino a 100 MHz.

Grazie a questo IC il bridge può assolvere tutte funzoni per cui è stato progettato, senza bisogno di particolari dispositivi esterni.

Ethernet PHY Il controller Ethernet interno al microcontrollore non dispone del layer fisico. Per tale motivo è stato necessario aggiungerlo esternamente attraverso il circuito integrato DP83848C della National Semiconductor.

E' questo l'integrato che va a collegarsi fra il microcontrollore e il connettore Ethernet RJ45.

Porta USB Il bridge è stato dotato di un connettore USB di tipo A collegato direttamente alla relativa porta del microcontrollore. Tramite l'interfaccia USB il bridge può essere conneso a delle memorie esterne, utili per il trasferimento di dati (come avviene nell'attuale versione del bridge UNOX).

Flash Memory La scheda è anche provvista di un chip di memoria Flash da 1 Mbit, la quale risulta direttamente connessa al microcontrollore; la memoria può servire per immagazzinare temporaneamente i dati ricevuti dal sistema di gestione remota.

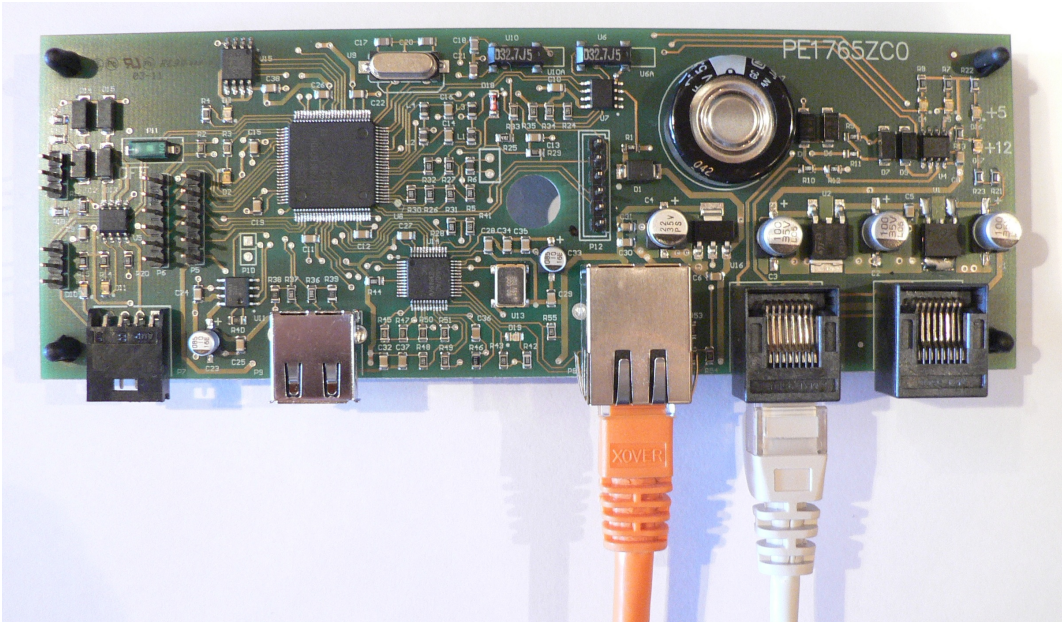

Figura 6.2: Foto del bridge Ethernet sviluppato da UNOX.

La scheda elettronica realizzata da UNOX è visibile in Figura 6.2. Nell'immagine sono visibili i cavi di collegamento al bus Modbus (cavo di colore grigio) e alla rete LAN tramite il cavo Ethernet (cavo di colore arancio). La porta USB-A per il collegamento delle memorie esterne è invece posizionata alla sinistra della porta Ethernet (connettore orizzontale argentato).

6.2 L'ambiente di sviluppo

L'ambiente di sviluppo per la scrittura del firmware interno al microcontrollore si compone di diversi moduli:

- IDE e editor di testo;
- compilatore;
- programmatore.

Considerando che UNOX negli anni ha acquisito un grande bagaglio di esperienza nella progettazione hardware e software di schede di controllo a microcontrollore, per la scrittura del firmware del nuovo bridge Ethernet l'azienda ha deciso di adottare l'ambiente di sviluppo utilizzato in tutti gli altri progetti più recenti.

Il linguaggio di programmazione scelto per la scrittura del codice sorgente è il C. Questa scelta è abbastanza comune nell'ambito dell'elettronica embedded, in quanto il linguaggio C è ormai da anni il linguaggio di riferimento per i programmatori di applicazioni a basso livello. Questa pratica comune ha portato, negli anni, allo sviluppo di una serie infinita di compilatori, IDE e librerie software che basano le proprie fondamenta proprio su questo linguaggio.

Anche la scelta relativa al sistema operativo utilizzato per lo sviluppo risulta alquanto scontata, poiché tutti i tool software utilizzati in ambito embedded sono sompatibili con sistemi operativi della famiglia Microsoft Windows.

L'IDE utilizzato per la programmazione del microcontrollore è Eclipse. Eclipse è un ambiente di sviluppo integrato multi-linguaggio e multipiattaforma. Ideato da un consorzio di grandi società quali Ericsson, HP, IBM, Intel, MontaVista Software, QNX, SAP e Serena Software, chiamato Eclipse Foundation, viene sviluppato da una comunità strutturata sullo stile dell'open source. Il programma è scritto in linguaggio Java, ma anziché basare la sua GUI su Swing, il toolkit grafico di Sun Microsystems, si appoggia a SWT, librerie di nuova concezione che conferiscono ad Eclipse un'elevata reattività. La piattaforma di sviluppo è incentrata sull'uso di plug-in, delle componenti software ideate per uno specifico scopo.

E proprio uno di questi plug-in viene utilizzato da UNOX per fare in modo che la procedura di compilazione possa essere avviata direttamente dall'IDE attraverso i suoi comandi standard. Infatti, l'azienda produttrice del compilatore C scelto mette a disposizione dei programmatori che utilizzano Eclipse un plug-in che, una volta installato, permette un completo interfacciamento del compilatore all'ambiente di sviluppo.

Il compilatore è prodotto dalla CodeSourcery e si chiama G++. Questo compilatore per C e C++ è disponibile in diverse versioni, in quanto supporta microprocessori con diverse architetture. Nel nostro caso la versione di riferimento quella compatibile con la famiglia di processori ARM. Il compilatore come output restituisce un file HEX, il classico formato ideato da Intel intorno al 1970.

Il file con il firmware scritto in formato HEX viene trasferito nella memoria Flash interna al microcontrollore attraverso un apposito tool di programmazione, composto da:

- convertitore USB-Seriale per PC;
- software di programmazione Flash Magic rilasciato dalla Embedded Systems Academy per microcontrollori NXP.

Il microcontrollore viene programmato via seriale sulla porta P0 (pin 98 e 99), ma per comodità UNOX utilizza di consueto un apposito convertitore USB-Seriale per evitare di utilizzare computer provvisti di porta seriale, oramai quasi introvabili. Il convertitore, una volta collegato alla porta USB del computer, richiede l'installazione dei propri driver in modo tale che nel sistema operativo la porta seriale venga riconosciuta come una classica porta COM.

6.3 Il SO in real-time FreeRTOS

Negli ultimi anni l'aumento delle funzioni messe a disposizione dalle apparecchiature elettroniche ha fatto sì che anche dispositivi di piccola taglia e con limitate risorse computazionali vengano equipaggiati con sistemi operativi in real-time.

Nell'ambito embedded la classica struttura ciclica di esecuzione serializzata di diverse procedure (contraddistinta dal noto ciclo primario *while(1)*) lascia sempre più spazio all'esecuzione di task paralleli controllati da un sistema operativo. Ad esempio, banalizzando, nella centralina di controllo di un'automobile possiamo avere il task dedicato al controllo del motore, il task dedicato all'impianto di climatizzazione, il task per in controllo dell'impianto airbag e così via.

Anche UNOX, considerate le sempre più crescenti esigenze della clientela e l'aumento della complessità elettronica dei propri prodotti, da diverso tempo sviluppa i propri firmware usando quest'ottica. Le diverse funzioni svolte da ogni forno vengono associate a task differenti e in controllo viene delegato al sistema operativo in real-time *FreeRTOS*[8].

FreeRTOS è un real-time operating system sviluppato specificatamente per dispositivi embedded, disponibile per diverse piattaforme di microcontrollori (fra le quali compare anche la famiglia ARM Cortex-M3). E' distribuito sotto licenza GPL ed il kernel consiste in soli 4 file scritti in linguaggio C; solo alcune parti di codice sono scritte in linguaggio assembly di basso livello.

La configurazione di FreeRTOS per l'utilizzo nel bridge di nuova concezione è stata effettuata da UNOX sulla base del bagaglio di esperienze già acquisito.

Per quanto riguarda l'organizzazione interna del codice sviluppato (e che verrà arricchito nelle fasi di sviluppo successive) possiamo identificare 3 task principali:

- task per la gestione del bus Modbus: il task gestisce le comunicazioni con gli altri dispositivi presenti in una colonna forni UNOX;
- task per la gestione dello Stack TCP/IP (che verrà presentato nel seguito del testo);
- task per la gestione dell'interfaccia USB 2.0.

6.4 Lo Stack TCP/IP lwIP

Nel bridge di UNOX l'implementazione della pila protocollare TCP/IP per la gestione delle comunicazioni su porta Ethernet viene destinata ad un particolare modulo software he prende il nome di *Stack TCP/IP*.

La diffusione sempre più capillare di dispositivi provvisti di interfaccia Ethernet ha fatto sì che negli ultimi anni diverse aziende e team di sviluppo di software open source abbiano rilasciato nel mercato svariate implementazioni di Stack TCP/IP. Ogni release integra funzioni più o meno specifiche, in base alla famiglia di microcontrollori per i quali viene sviluppata ed in base alle funzionalità richieste dal progetto. In molti casi sono le stesse aziende produttrici di microcontrollori provvisti di controller Ethernet a rilasciare gratuitamente ai propri clienti implementazioni proprietarie dei protocolli TCP/IP.

UNOX, in collaborazione con il CNR, ha scelto di implementare la gestione delle comunicazioni su interfaccia Ethernet con lo *Stack TCP/IP lwIP* (Lightweight TCP/IP)[9]. Dato che il codice sorgente di questo stack viene rilasciato sotto un'implementazione generica, il gruppo di lavoro del CNR, sotto la guida del Prof. Ivan Cibrario Bertolotti, ha eseguito il porting dell'intero progetto per renderlo pienamente funzionante con il microcontrollore di NXP scelto.

Originariamente scritto da Adam Dunkels del Swedish Institute of Computer Science, lwIP viene ora sviluppato da un team di programmatori distribuiti in tutto il mondo e coordinati da Leon Woestenberg.

Fin dal suo primo rilascio lwIP ha riscosso un notevole successo e oggigiorno è usato in molti dispositivi commerciali. La struttura dello stack permette, inoltre, di usarlo anche in dispositivi che integrano un SO in real-time (proprio come nel caso del bridge UNOX).

L'obiettivo dell'implementazione adottata da lwIP sta nell'esigua occupazione di memeoria RAM a fronte di una completa realizzazione del protocollo TCP; questo permette allo stack di consumare qualche decina di kbyte di RAM e circa 40 kbyte di memoria ROM (Flash).

I principali protocolli supportati da lwIP sono i seguenti: IP, ICMP, UDP, TCP, DHCP, PPP e ARP.

6.4.1 Configurazione e inizializzazione dello stack

In questa sezione verranno analizzate le più importanti direttive di configurazione dello stack e le funzioni dedicate all'inizializzazione della pila protocollare.

Per prima cosa va evidenziato che nel progetto messo a punto dal CNR tutte le direttive *#define* di configurazione dello Stack sono collocate nel file *lwipopts.h*. Ogni protocollo può essere abilitato impostando correttamente la relativa direttiva di abilitazione. Ad esempio, per attivare il protocollo ARP e disabilitare l'ICMP è necessario inserire le seguenti righe di codice:

```
1 #define LWIP_ARP  1
2 #define LWIP_ICMP 0
```

Oltre all'abilitazione dei vari protocolli lo stack mette a disposizione una serie di leve che consentono di adattare ogni modulo alle proprie esigenze.

Le principali direttive di configurazione sono le seguenti:

- MEM_ALIGNMENT: deve essere impostata in modo tale da rispecchiare l'architettura della CPU (in byte); nel caso dell'LPC1768 va impostata a 4 (32 bit);
- MEM_SIZE: numero di byte dedicato alla heap-memory;
- MEMP_NUM_UDP_PCB: numero degli UDP control block - uno per ogni connessione UDP attiva;
- MEMP_NUM_TCP_PCB: numero massimo delle connessioni TCP gestite simultaneamente;
- MEMP_NUM_TCP_PCB_LISTEN: numero massimo delle connessioni TCP in ascolto;
- MEMP_NUM_TCP_SEG: numero di segmenti TCP accodati simultaneamente.

Passiamo ora all'analisi delle funzioni relative all'inizializzazione dello stack.

Innanzitutto, prima di invocare qualsiasi altra funzione di lwIP, occorre richiamare l'apposita funzione di inzializzazione nel modo seguente:

```
1 tcpip_init(NULL,NULL);
```

I due parametri di ingresso della funzione *tcpip_init* consentono di specificare il nome di una funzione (e i relativi parametri di ingresso) che deve essere richiamata dopo che la fase di inzializzazione dello stack è terminata.

Attraverso la funzione *sys_thread_new* di FreeRTOS è invece possibile creare il task relativo allo stack lwIP e successivamente avviarlo tramite la funzione *vTaskStartScheduler*:

```
1 sys_thread_new((char *)"TCPIP", tcp_thread, NULL, TCPIP_STACKSIZE,
      TCPIP_TASK_PRIORITY);
2 /* Creazione di altri task */
3 vTaskStartScheduler();
```

Alla funzione *sys_thread_new* occorre passare il nome dedicato al task, il nome della funzione da eseguire, i suoi parametri di ingresso, la dimensione dell'area di stack riservata alla funzione e la priorità assegnata al task.

6.4.2 Applicazione di test: scrittura dei programmi di cottura

Come vedremo in seguito durante l'attività finale di tirocinio sono stati eseguiti dei test per callaudare l'intero sistema e per ottenere alcuni dati che esprimano la bontà del lavoro svolto.

Per eseguire i test è stato scritto un apposito firmware in grado di emulare la procedura di scrittura dei programmi di cottura. In sostanza, una volta ricevuta la richiesta via TCP da un determinato nodo server, il programma risponde con l'XML *setProgrCottT* con status uguale a complete, come se la richiesta fosse stata effettivamente eseguita con successo.

Analizziamo dunque il codice del task utilizzato per eseguire i test sullo Stack lwIP:

```
1  /* strCopy - Funzione accessoria */
2  // Copia la stringa sorg in dest (senza fine stringa)
3  void strCopy(char *dest, char *sorg)
4  {
5    while (*sorg != 0)
6    {
7      *dest = *sorg;
8      dest++;
9      sorg++;
10   }
11 }
12
13 /* tcp_thread */
14 // Task di gestione delle connessioni TCP
15 void tcp_thread(void *arg)
16 {
17   struct netconn *conn, *newconn;
18   struct netif lpc17xx_netif;
19   struct ip_addr IpAddr, NetMask, GateWay;
20   struct netbuf *buf;
21   char *rxString, *actKeyStart, *actKeyStop;
22   unsigned short rxStringLength;
23   char resp1[] = "<setProgrCottT><request>";
24   char resp2[] = "</request><response><status>complete</status></
       response></setProgrCottT>";
25   char actKeyTagOpen[] = "<key>";
26   char actKeyTagClose[] = "</key>";
27   char outStr[140];
28   unsigned char cont;
29
30   /* Setting degli indirizzi IP e delle interfacce di rete */
31   IP4_ADDR(&IpAddr, 192, 168, 0, 65);
32   IP4_ADDR(&NetMask, 255, 255, 255 ,0);
33   IP4_ADDR(&GateWay, 192, 168, 0, 1);
34   _DBG_("IP, NETMASK and GW initialized!\n");
35   netif_add(&lpc17xx_netif, &IpAddr, &NetMask, &GateWay, NULL,
       lpc17xx_netif_init, tcpip_input);
36   _DBG_("netif_add!\n");
37   netif_set_default(&lpc17xx_netif);
38   _DBG_("netif_set_default!\n");
```

```
39    netif_set_up(&lpc17xx_netif);
40    _DBG_("Network␣interface␣initialized!\n");
41
42    LWIP_UNUSED_ARG(arg);
43    /* Crea un nuovo identificativo di connessione */
44    conn = netconn_new(NETCONN_TCP);
45    /* Bind della connessione sulla porta 5065 */
46    netconn_bind(conn, NULL, 5065);
47    /* La connessione si mette in ascolto... */
48    netconn_listen(conn);
49
50    while (1)
51    {
52      /* Prelievo della nuova connessione */
53      newconn = netconn_accept(conn);
54
55      /* Processo la nuova connessione */
56      buf = netconn_recv(newconn);
57      if(buf != NULL)
58      {
59        netbuf_data(buf, (void *)&rxString, &rxStringLength);
60        actKeyStart = strstr(rxString, actKeyTagOpen);
61        actKeyStop = strstr(rxString, actKeyTagClose);
62        *actKeyStop = 0;
63        cont = 0;
64        strCopy(outStr, resp1);
65        cont += 24;
66        strCopy(outStr+cont, actKeyTagOpen);
67        cont += 5;
68        strCopy(outStr+cont, actKeyStart+5);
69        cont += actKeyStop-actKeyStart-5;
70        strCopy(outStr+cont, actKeyTagClose);
71        cont += 6;
72        strCopy(outStr+cont, resp2);
73        cont += 72;
74        netconn_write(newconn, outStr, cont, NETCONN_COPY);
75        netbuf_delete(buf);
76        _DBG_("Send␣TCP␣message␣from␣server.\n");
77      }
78      else
79        _DBG_("Connection␣closed␣by␣client.\n");
80
81      /* Chiusura della connessione */
82      netconn_close(newconn);
83      netconn_delete(newconn);
84    }
85  }
```

Nelle prime righe di codice della funzione *tcp_thread* (task di gestione delle comunicazioni TCP) tramite la funzione *netif_add* viene aggiunta allo stack l'interfaccia di rete Ethernet, utilizzando gli indirizzi di rete specificati.

In seguito, tramite l'invocazione delle funzioni *netif_set_default* e *netif_set_up* la porta Ethernet viene configurata come interfaccia di default e poi abilitata.

Con il blocco di istruzioni successive il task genera un nuovo identificativo di connessione e si mette in ascolto sulla porta TCP 5065.

All'interno del ciclo *while(1)*, all'arrivo di una nuova connessione, tramite la funzione *netbuf_ data* viene prelavata la stringa di dati ricevuta, il cui riferimento viene copiato in *rxString*. A questo punto, prima di rispondere al nodo server che l'operazione è stata eseguita con successo, occorre prelevare la chiave attribuita all'attività, in quanto l'XML di risposta *setProgrCottT* in base alle specifiche analizzate deve riportare tale riferimento.

Per tale motivo attraverso l'invocazione della funzione *strstr* vengono prelevati i puntatori di inzio e fine chiave. Ciò che viene fatto è cercare all'interno della stringa *rxString* le sotto-stringhe *<key>* e *</key>*.

Una volta ottenuti i riferimenti si procede alla composizione della risposta, copiando il risultato in *outStr*. Una volta che la risposta è pronta viene inviata tramite la funzione *netconn_ write*. Infine, terminato l'invio, la connessione viene chiusa e tutti i buffer vengono liberati.

Dunque, come si è visto il codice risulta alquanto semplice e pur essendo un programma di test rispecchia comunque la struttura base delle routine più complesse di gestione delle attività. Chiaramente il codice che verrà scritto nelle successive fasi di sviluppo dovrà integrare parser e altre routine di interpretazione e costruzione degli XML, ma la gestione delle connessioni seguirà lo schema di base appena analizzato.

Capitolo 7

Collaudo del sistema e conclusioni

Le fasi di collaudo sono state eseguite attraverso dei test compiuti sulle singole parti e sull'intero sistema. Dapprima è stato verificato il corretto funzionamento del servizio di sistema, successivamente quello di interfaccia utente e bridge. Infine i diversi moduli sono stati installati in un sistema composto da 10 nodi server e da 6 forni per verificarne il funzionamento globale.

Le verifiche che sono state eseguite hanno permesso di accertare che il lavoro è stato svolto correttamente e che il sistema ideato, oltre che rispondere alle esigenze analizzate, risulta robusto e performante.

7.1 Collaudo del servizio di sistema

I test compiuti sull'applicativo sono stati molteplici. Le prove sono state eseguite direttamente durante le fasi di sviluppo seguendo questa procedura:

- avviando 4-5 nodi che utilizzano altrettanti database, tutti residenti sullo stesso server MySQL; ogni nodo esegue una copia compilata del software ed è configurato per utilizzare un'apposita porta TCP;
- avviando un nodo (come root dell'albero) in modalità *debug* di MS Visual Studio, in modo da rilevare eventuali problemi sul codice;
- avviando un software di test che simula il funzionamento del bridge installato sui forni.

Tutti i nodi avviati vengono eseguiti sullo stesso computer; per tale ragione tutte le comunicazioni TCP avvengono in *localhost* (indirizzo IP 127.0.0.1).

Per testare il funzionamento del sistema così strutturato, si è fatto ricorso all'applicativo *URCTestClient*, appositamente creato per eseguire le prove di collaudo. Il programma permette di inviare via TCP ad un particolare nodo server (solitamente il nodo radice) una determinata attività, come se questa gli venisse trasmessa da un nodo di livello superiore. L'invio può essere eseguito manualmente o automaticamente attraverso l'utilizzo di un timer programmato a cadenza regolare.

7.2 Webapp di interfaccia utente

I test compiuti sull'applicativo di interfaccia utente si dividono in tre tipi:

- test di compatibilità cross-browser;
- test di usabilità;
- test di compatibilità con il servizio di sistema.

I test di compatibilità sono stati eseguiti su browser differenti in architetture differenti. La tabella 7.1 riassume i test eseguiti. La dicitura "*n.d.*" indica che il browser

	Chrome	Firefox	MS IE	Safari	Opera
MS Windows	SI	SI	SI (fino v. 5)	SI	SI
Linux Ubuntu	SI	SI	n.d.	n.d.	—
Apple MacOS X	SI	SI	n.d.	SI	SI
Apple iOS	n.d.	n.d.	n.d.	SI	—

Tabella 7.1: Compatibilità della webapp su piattaforme e browser differenti.

non esiste per quella specifica versione, mentre il simbolo "—" indica che il test non è stato effettuato.

Le prove sono state eseguite durante l'intera fase di sviluppo dell'applicativo, in modo da correggere immediatamente eventuali errori nel codice.

Nella webapp realizzata la compatibilità di AJAX e jQuery è garantita proprio per le caratteristiche intrinseche delle due librerie. Ciò che ha richiesto molto più tempo è stato il debug dell'impaginazione sul codice HTML e CSS.

In sintesi l'applicazione risulta comunque compatibile con tutti i più popolari browser e piattaforme presenti in commercio.

Per quanto riguarda le prove di usabilità si è chiesto a 8 diversi utenti di utilizzare l'applicativo dopo aver spiegato loro a cosa servisse. Gli utenti che hanno partecipato al test possiedono differenti livelli di conoscenza in ambito informatico (2 soli utenti possiedono buone conoscenze).

Le impressioni raccolte dagli 8 utilizzatori sono state positive, sia in termini di usabilità che in termini di apprezzamento grafico. L'interfaccia non ha rappresentato in nessun caso un ostacolo all'utilizzo del sistema e le consegne che sono state affidate loro sono state svolte velocemente e con successo.

Altri test che hanno interessato la webapp di interfaccia hanno riguardato la compatibilità con il servizio di sistema, ma in tal caso è stato sufficiente garantire il rispetto delle specifiche XML per assicurare la piena compatibilità dei due software.

7.3 Collaudo dello Stack lwIP

La quasi totalità dei test eseguiti sullo stack TCP/IP sono stati svolti dal CNR durante le fasi di porting del software.

I test hanno riguardato, oltre che il collaudo dell'intero modulo, la compatibilità con il sistema operativo FreeRTOS e gli indici per l'analisi del throughput. Proprio su quest'ultimo punto è stato svolto un intenso lavoro di ottimizzazione del codice, al fine di rendere lo stack altamente performante.

Durante l'attività di tirocinio, una volta acquisito il lavoro svolto dal CNR, si sono svolti dei test su dei semplici task di gestione delle connessioni TCP per la realizzazione dell'echo dei dati ricevuti.

Inoltre sono stati avviati dei test di ping per verificare l'effettiva tenuta dello stack e dell'hardware. Di seguito viene riportato l'esito di un test di ping:

```
1 --- 192.168.0.65 ping statistics ---
2 12422 packets transmitted, 12422 packets received, 0.0% packet loss
3 round-trip min/avg/max/stddev = 0.879/5.425/81.424/6.036 ms
```

I risultati evidenziano come in media le richieste vengano evase in 5,4 ms e come la gestione dello stack da parte del sistema operativo non causi alcuna perdita dei dati e delle connessioni TCP.

7.4 Collaudo del sistema nella scrittura dei programmi di cottura

Il collaudo del sistema realizzato in base alle direttive fin qui analizzate è stato effettuato predisponendo 3 computer e 1 bridge Ethernet. I 4 dispositivi sono stati posizionati in 2 sotto-rete LAN indipendenti e le comunicazioni sono avvenute tramite due diverse connessioni ad Internet.

Il sistema URC di collaudo è stato configurato come evidenziato in Figura 7.1. I blocchi di colori diversi identificano i dispositivi utilizzati: in arancio, azzurro e viola i tre computer, in verde il bridge.

L'albero così strutturato presenta nodi con altezza massima di 4 livelli e conta 6 nodi foglia (forni). In realtà, però, come illustrato in figura, i 6 forni sono stati emulati utilizzando lo stesso bridge Ethernet, così da appesantire ulteriormente il lavoro a cui sono sottoposti il SO FreeRTOS e lo Stack lwIP.

Le caratteristiche dei computer e del bridge utilizzati per eseguire i test sono riassunte in Tabella 7.2. La tabella evidenzia che il PC 1 e il bridge sono stati installati nella prima rete LAN con connessione ad Internet n. 1, mentre PC 2 e 3 sono stati installati in una seconda rete con connessione ad Internet n. 2. Le caratteristiche delle due connessioni ad Internet sono le seguenti:

1. Ponte radio su standard Hiperlan, con ping medio al MIX[1] di 15-20 ms;
2. ADSL con ping medio al MIX di 20-30 ms.

Inoltre, per analizzare la stabilità del collegamento ad Internet durante l'intera durata della prova è stato avviato un test di ping fra il PC 1 e il modem ADSL della connessione n. 2. I risultati del test sono i seguenti:

[1] MIX è l'acronimo di Milan Internet eXchange, ovvero il più importante punto di interscambio tra Internet Service Provider in Italia.

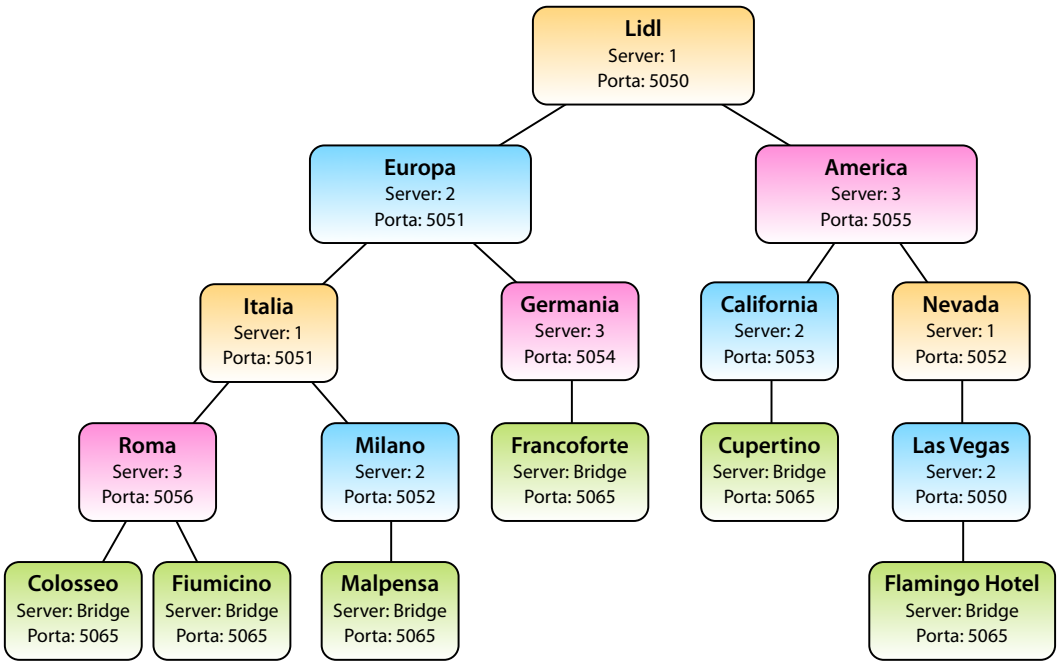

Figura 7.1: Diagramma ad albero dei nodi del sistema di collaudo.

- N. pacchetti inviati: 35.963
- N. pacchetti persi: 179
- Durata media di una comunicazione: 55 ms

In sostanza, analizzando la struttura ad albero di Figura 7.1 si nota come tutte le connessioni avvengano tramite Internet (grazie di un interscambio di connessione fra nodi di livelli adiacenti), tranne quelle fra i nodi (Europa, Germania) e (America, California).

Per eseguire il test si è fatto ricorso al programma di collaudo *URCTestClient*, che è stato programmato per inviare un'attività di scrittura di un programma di cottura al nodo Lidl (radice dell'albero) ogni 3 secondi.

Ogni attività è stata indirizzata ciclicamente ad uno dei nodi forno, in modo da coinvolgere nella continua attività di programmazione tutti i nodi dell'albero. *URCTestClient* è stato eseguito sul *PC 2*, pertanto anche il caricamento di ogni attività madre è stato fatto sfruttando la rete Internet.

Va specificato che il modulo client del servizio di sistema di tutti i nodi server è stato cofigurato per avere un tick del timer di esecuzione delle attività di 500 ms.

I risultati che sono stati ottenuti dal nodo Lidl a test concluso sono i seguenti:

- Durata del test: più di 10 ore
- N. totale attività avviate: 12.060
- N. attività non completate: 0

Disp.	Hardware	SO	Conn. Internet	Nodi
PC 1	iMac 21,5" Intel 3,06 GHz (single core) 1 GB RAM	MS Windows 7 Pro Config. in Virtual Machine	1	Lidl, Italia, Nevada
PC 2	MacBookPro Intel 2,5 GHz (single core) 512 MB RAM	MS Windows XP Pro Config. in Virtual Machine	2	Las Vegas, Europa, Milano, California
PC 3	Notebook Acer Intel 1 GHz 256 MB RAM	MS Windows XP Pro	2	Germania, America, Roma
Bridge	—	FreeRTOS	1	Tutti i nodi forno

Tabella 7.2: Caratteristiche dei dispositivi (computer e bridge) utilizzati nel sistema di collaudo.

- N. attività con doppio tentativo di invio: 10
- N. attività con triplo tentativo di invio: 0
- Durata media esecuzione attività: 2,3086 secondi
- Attività con durata maggiore: 23 secondi

I risultati del test sono estremamente positivi in quanto, come si evidenzia nei dati raccolti, il sistema ha eseguito tutte le attività avviate.

I diversi nodi server, che sono stati installati in macchine con limitate capacità computazionali, hanno retto al grande carico di lavoro a cui sono stati sottoposti e i tempi di esecuzione delle attività sono più che buoni.

Le sole 10 ri-trasmissioni che sono state effettuate sul nodo Lidl sono sicuramente correlate ai test di ping ottenuti, i quali evidenziano 179 pacchetti persi. Tale ipotesi viene avvalorata dal fatto che lo stesso test è stato ripetuto posizionando tutti i dispositivi (computer e bridge) all'interno della stessa rete LAN e in questo caso non si è avuta nemmeno una ritrasmissione (con più di 10.265 attività avviate e il 100% di attività compiute).

Altri importanti risultati si ottengono analizzando il tempo medio di esecuzione delle attività relazionato alla profondità dei diversi nodi. La Tabella 7.3 evidenzia i dati che sono stati ricavati dal database del nodo Lidl.

Come è visibile il tempo medio di esecuzione di un'attività indirizzata ad uno specifico nodo aumenta con la profondità in cui si trova il nodo stesso. Tale risultato, oltre a confermare le aspettative, indica che i test sono stati compiuti regolarmente.

Nome nodo	Livello nodo	Durata media esec. att. [sec]
Colosseo	4	2,5743
Fiumicino	4	2,6185
Malpensa	4	2,5141
Francoforte	3	1,8353
Cupertino	3	1,8434
Flamingo Hotel	4	2,4659

Tabella 7.3: Tempi medi di esecuzione delle attività ottenuti durante il test di collaudo.

7.5 Conclusioni e sviluppi futuri

La presentazione dello studio che ha portato alla progettazione del sistema URC di UNOX è terminata.

Gli obiettivi visti alla sezione 1.3 del testo sono stati portati a termine tutti con successo.

I test preliminari che sono stati svolti durante le fasi conclusive del lavoro hanno dimostrato che il sistema ideato risulta robusto e performante, anche in sistemi con più livelli e sottoposti ad un continuo carico di lavoro. Chiaramente per poter simulare con maggior precisione un caso reale occorrerà eseguire nuovi test, gestendo molti più nodi e l'invio di richieste simultanee.

Gli sviluppi futuri del progetto prevedono pertanto lo svolgimento delle seguenti attività:

- implementazione delle routine relative alle altre attività (import/export di log, parametri di funzionamento, ecc);
- integrazione delle funzionalità offerte dall'applicativo di interfaccia utente (come conseguenza del punto precedente);
- sviluppo del firmware per il bridge Ethernet;
- ulteriori collaudi del servizio di sistema con framework .NET in ambienti diversi da MS Windows (ad esempio con l'utilizzo di Mono);
- ulteriori test e collaudi dell'intero sistema.

Bibliografia

[1] IDC, International Data Corporation, http://www.idc.com/getdoc.jsp?containerId=prUS22360110

[2] Nagel C., et alii, *C# 2008 – Guida per lo sviluppatore*, Hoepli, 2008

[3] Wankyu Choi, et alii, *PHP4 - Guida per lo sviluppatore*, Hoepli, 2003

[4] Massimo Canducci, *PHP 5*, Apogeo, 2004

[5] jQuery, http://jquery.com/

[6] Zakas Nicholas, *AJAX - Guida per lo sviluppatore*, Hoepli, 2006

[7] H. M. Deitel, P. J. Deitel, *C++ Fondamenti di programmazione*, Apogeo, 2005

[8] Richard Barry, *Using the FreeRTOS Real Time Kernel - a Practical Guide - NXP LPC17xx Edition*, 2010

[9] Adam Dunkels, *Design and Implementation of the lwIP TCP/IP Stack*, Swedish Institute of Computer Science, February 20, 2001