ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA IN INGEGNERIA ELETTRONICA E TELECOMUNICAZIONI

TITOLO DELL'ELABORATO

IDENTIFICAZIONE DI UTENTI E OGGETTI IN AMBIENTI INTELLIGENTI MEDIANTE TECNICHE DI COMPUTER VISION

Elaborato in
CALCOLATORI ELETTRONICI

Relatore
Luca Roffia

Presentata da
Luca Feltrin

Sessione II°
Anno Accademico 2010/2011

Indice

1 Abstract

L'obiettivo di questo studio è capire come identificare oggetti o persone mediante tecniche di computer vision.

La computer vision è la scienza che permette a macchine e calcolatori di “vedere”, ovvero estrarre informazioni dalle immagini di una telecamera, per esempio, ed usarle come input dall'ambiente per le proprie computazioni.

Al di là dell'identificazione di oggetti, gli utilizzi di queste tecniche sono innumerevoli: riconoscimento di movimenti di parti del corpo di un utente (mani, busto, occhi, direzione del viso) per controllare un cursore, tracking di oggetti veloci come palle da ping pong, in ambito industriale sono usate per la valutazione della qualità di un prodotto, in ambito biomedicale per analizzare immagini mediche, oppure per rilevare eventi come l'ingresso di un utente in un ambiente per la videosorveglianza.

Il problema dell'identificazione di oggetti viene risolto in due fasi, l'individuazione e il riconoscimento.

Nella prima fase, la macchina data un'immagine deve essere in grado di riconoscere che all'interno di essa c'è un oggetto della categoria specificata e quindi individuarne posizione e la porzione di immagine che lo contiene.

Gli algoritmi spesso possono essere “addestrati” per riconoscere una certa categoria di oggetti quindi ipoteticamente può essere riconosciuta qualsiasi cosa.

L'algoritmo più usato e semplice è quello delle Haar-like features [1] ed esistono dei file che contengono già le informazioni su una certa categoria di oggetti ottenute in seguito alla fase di learning.

Una volta individuato l'oggetto all'interno dell'immagine si procede al riconoscimento dato un database di immagini: l'algoritmo si limiterà a decidere quale immagine del database è la più simile a quella data in ingresso.

Un altro modo per identificare qualcosa è banalmente associare fisicamente un identificatore univoco all'oggetto come per esempio un tag RFID o un codice a barre, quindi una volta riconosciuto questo identificatore è riconosciuto automaticamente anche l'oggetto. Mediante la Computer Vision è possibile riconoscere codici a barre monodimensionali o bidimensionali come i QRCode [2].

2 Teoria Generale

2.1 Codici a Barre

I codici a barre sono un insieme di elementi grafici a contrasto elevato disposti in modo da poter essere letti da un sensore a scansione e decodificati per restituire l'informazione contenuta. [3]

2.1.1 Cenni sui Codici a Barre

I codici a barre furono teorizzati nel 1948 e utilizzati per la prima volta nel 1973 con lo scopo di identificare univocamente i prodotti presenti nei magazzini e per automatizzare le operazioni di cassa.

L'idea era quella di riuscire a codificare delle informazioni in un'immagine che potesse essere letta facilmente da uno scanner ottico, l'altro contrasto di queste immagini, in genere composte da bande o quadrati bianchi e neri, serve a rendere più facile la lettura.

L'utilizzo nativo è ormai routine quotidiana nei supermercati, dove lettori a laser o a sensore CCD riescono a leggere il codice in tempi minimi, inoltre al giorno d'oggi vengono utilizzati in un'enorme quantità di applicazioni in cui c'è la necessità di etichettare o identificare oggetti in generale, si può dire che i codici a barre sono gli antenati della tecnologia RFID che grazie all'utilizzo di microprocessori permette di codificare un maggior numero di informazioni che possono anche cambiare nel tempo.

Ciononostante i codici a barre restano il metodo di identificazione più veloce ed economico in quanto il lettore può essere la fotocamera presente in un normale cellulare che grazie ad alcuni algoritmi di scansione è in grado di riconoscere e decodificare il codice in tempi comunque brevi, e il tag può essere un semplice foglio di carta.

Inoltre i codici a barre, in genere, incorporano un meccanismo di controllo e correzione degli errori a ridondanza

che permette di decodificare l'informazione anche nel caso in cui l'immagine sia parzialmente danneggiata.

Esistono due categorie principali di codici a barre: i codici 1D o lineari e i codici 2D o a matrice.

I primi sono quelli più conosciuti sin dagli anni 70/80, la sigla “1D” deriva dal fatto che l'immagine si sviluppa in una sola dimensione con un susseguirsi, solitamente, di barre nere più o meno spesse su sfondo bianco.

Figura 2.1: Esempio di codice a barre lineare: EAN-13

I più importanti tipi di codici a barre lineari sono l'UPC (Universal Product Code) usato principalmente nel nord America e nel regno unito, la sua controparte europea, l'EAN *(Figura 2.1)* (European Article Number ora rinominato International Article Number in quanto usato anche in Giappone) e il Pharmacode usato per i prodotti farmaceutici.

I codici lineari tuttavia non hanno un buon rapporto tra l'informazione contenuta e la superficie occupata dall'immagine, al massimo gli standard in circolazione possono codificare al massimo numeri con circa 20 cifre (circa 66 bit), numeri più grandi richiederebbero codici a barre troppo lunghi per poter essere letti con strumenti economici.


Figura 2.2: Esempio di codice a matrice: AztecCode

I codici 2D o a matrice risolvono questo problema, in quanto sviluppandosi in due dimensioni, riescono a contenere molte più informazioni a parità di superficie occupata dall'immagine.

Grazie alla potenza di calcolo raggiunta dai dispositivi odierni dotati di fotocamera un codice bidimensionale può essere riconosciuto e decodificato in tempi paragonabili a quelli dei codici lineari.

Il più conosciuto è senza dubbio il QRCode, dove QR sta per Quick Responce, progettato appositamente per essere letto

nel minor tempo possibile, questo codice sarà oggetto di studio in questa tesi.

Altri codici usati comunemente sono il DataMatrix e l'Aztec Code *(Figura 2.2)*, entrambi simili al QRCode.

Altri codici interessanti sono il Bokode che consiste in un led coperto da una maschera contenente un codice DataMatrix di dimensioni estremamente ridotte (il diametro del led stesso, ovvero circa 3mm) che può essere letto da una fotocamera regolando la messa a fuoco all'infinito, e l'HCCB *(Figura 2.3)* (High Capacity Color Barcode) della Microsoft che usa anche i colori (Nero Giallo Magenta e Cyano) per aumentare ulteriormente la quantità di informazioni codificata per unità di superficie.

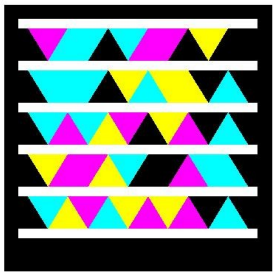


Figura 2.3: Esempio di HCCB

Un esempio di codice “ibrido” è il PDF417 detto di tipo “Stacked Bar” che consiste in una serie di codici a barre lineari posti uno sopra l'altro.

2.1.2 QRCode

Il QRCode fu inventato nel 1994 dalla Denso-Wave, sussidiaria della Toyota con l'intento di creare un codice che contenesse un gran numero di informazioni anche di tipi diversi e che fosse velocemente leggibile.

Le caratteristiche principali di questo codice sono:

- l'alto numero di dati che può contenere: numeri fino a 7089 cifre o 4296 caratteri testuali (Tabella 2.1).
- Capacità di contenere Kanji e Kana (gli ideogrammi orientali)
- Capacità di correzione degli errori fino ad un massimo del 30% grazie all'algoritmo Reed-Solomon, ci sono 4 livelli (L M Q H) che regolano il rapporto tra il payload e i caratteri ridondanti per la correzione.

- Può essere letto a 360° grazie ai 3 quadrati presenti negli angoli superiori e a sinistra che servono appunto a rilevare l'allineamento.
- Sono uno standard Free in quanto i diritti posseduti dalla Denso Wave non vengono esercitati.

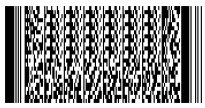
		QR Code	PDF417	DataMatrix	Maxi Code
					
Developer(country)		DENSO(Japan)	Symbol Technologies (USA)	RVSI Acuity CiMatrix (USA)	UPS (USA)
Type		Matrix	Stacked Bar Code	Matrix	Matrix
	Numeric	7,089	2,710	3,116	138
	Alphanumeric	4,296	1,850	2,355	93
	Binary	2,953	1,018	1,556	
	Kanji	1,817	554	778	
Standardization		AIM International JIS ISO	AIM International ISO	AIM International ISO	AIM International ISO

Tabella 2.1: Confronto tra QRCode e vari tipi di codici a matrice

Lo standard è stato definito nel giugno del 2000 sotto la sigla ISO/IEC18004, mentre nel 2004 è stata definita anche la

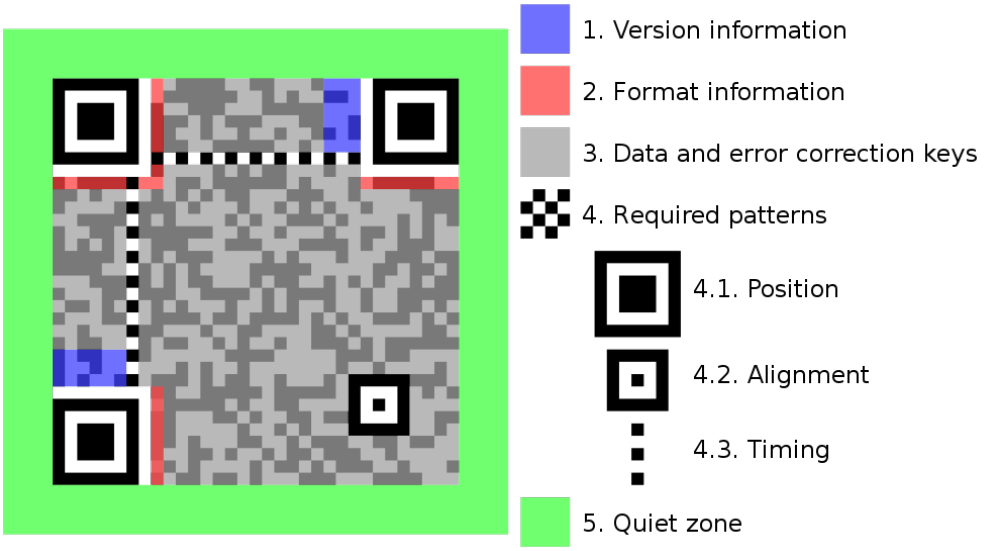


Figura 2.4: Struttura di un QRCode

versione “Micro QRCode” che dovrebbe rendere il codice ancora più piccolo a discapito del payload e della correzione degli errori.

Con il termine modulo si intende ogni “punto” minimo che forma la figura e che può essere o nero o bianco.

La versione di un codice indica da quanti moduli è composta, ovvero le dimensioni stesse, da un minimo di 21x21 moduli per la Versione 1 fino a un massimo di 177x177 moduli per la Versione 40.

Man mano che il codice aumenta di dimensioni sono necessarie delle figure note per l'allineamento in posizioni regolari simili alle tre situate negli angoli.

Nell'immagine sono presenti anche le informazioni sulla codifica dei caratteri contenuti all'interno ovvero numerico, testo, Kanji o binario.

Si fa notare che anche per il formato “numerico” o “binario” i dati sono sempre sotto forma di testo, ma cambia la codifica dei caratteri, nel caso del numerico si utilizzano 10 bit ogni 3 cifre in quanto ci sono solo 10 simboli da codificare, nel caso binario ogni carattere è codificato secondo il set Latin/Kana a 8 bit per carattere.[4]

Da notare che visto che i QRCode vengono utilizzati soprattutto negli smartphone si vogliono condividere informazioni di tipo diverso in modo semplice e veloce, quindi si sono sviluppati nuovi formati che consistono semplicemente in una sintassi da rispettare per il payload che possono essere SMS, contatti telefonici, coordinate geografiche e in teoria qualsiasi altro formato che chiunque potrebbe inventare.

- Nell'esempio in Figura 2.5 è stata codificata la posizione geografica della sede principale di ingegneria a

Figura 2.5: Esempio di posizione Geografica (Via Genova 181, Cesena)

cesena in via genova 181, ma il QRCode è di tipo testuale e contiene semplicemente “geo:latitudine: longitudine”, ma un comune applicazione per la lettura di QRCode è in grado di riconoscerla come posizione e quindi di mostrare, per esempio, il comando “Visualizza sulla mappa”.[5][6]

2.2 Computer Vision

La computer vision, o machine vision, è la scienza che studia come ricavare informazioni da immagini o filmati.

È una tecnologia che si sta sviluppando solo recentemente grazie anche alle potenze di calcolo raggiunte dai dispositivi sia fissi che mobili.

Le applicazioni sono innumerevoli, dall'ambito industriale per controllare i movimenti di robot, per rilevare la qualità o il semplice passaggio di oggetti in una linea di produzione, in ambito biomedicale per studiare radiografie e altri tipi di immagini relative a un paziente o per realizzare nuove forme di interazione con le macchine.

L'insieme di tutti i problemi legati alla computer vision si possono riassumere in alcune categorie:

- **Recognition**, ovvero riconoscere oggetti o caratteristiche particolari all'interno dell'immagine come per esempio un volto, una figura geometrica, o dei caratteri stampati.
  Purtroppo per risolvere questo problema esistono una moltitudine di algoritmi dedicati a specifici compiti e a specifiche situazioni ed è complicato e tutt'ora oggetto di studio formalizzare meglio il problema per situazioni arbitrarie.
  La recognition a sua volta può essere divisa in tre categorie: l'**Object Recognition** ovvero ricondurre un oggetto all'interno dell'immagine a una categoria più ampia, per esempio data l'immagine di uno sgabello può essere ricondotta alla categoria “posti dove sedersi”, l'**Identification** ovvero il riconoscimento di uno specifico

oggetto, per esempio il volto di una persona, e infine la **Detection** ovvero il riconoscimento di una certa caratteristica o tipologia di oggetto all'interno dell'immagine, per esempio riconoscere la presenza di uno o più volti oppure la presenza di cellule infette in immagini mediche.
In genere per risolvere un problema possono essere necessari più passaggi, per esempio per l'identificazione di un volto bisogna prima eseguire la Detection e solo dopo l'Identification.

- **Motion Analysis** ovvero la stima del moto di oggetti. Anche in questo caso esistono più categorie: **Egomotion** ovvero stimare la posizione e la rotazione della camera in base al filmato che sta riprendendo, il **Tracking** ovvero seguire i movimenti di uno o più oggetti all'interno della scena, per esempio una pallina da ping pong o più persone ed infine l'**Optical Flow** ovvero stimare il movimento di ogni punto dell'immagine rispetto allo schermo stesso, può essere usato per riconoscere i gesti di un utente che vanno interpretati come comandi.
- **Scene Reconstruction** ovvero costruire un modello 3D dello scenario in cui la camera da presa si sta muovendo.
- **Image Restoration** ovvero la ricostruzione di un'immagine danneggiata o semplicemente immersa nel rumore in base alle caratteristiche delle zone circostanti a quella danneggiata. Un esempio di applicazione è l'"Inpainting".[7]

In generale quando si vuole eseguire un qualche tipo di analisi è necessario preparare l'immagine in input con una fase di pre-processing con la quale si evidenziano le caratteristiche che si vogliono rilevare e si attenuano tutti gli elementi che possono falsare il risultato finale come il rumore. Questa fase dipende fortemente dal problema da risolvere. [8]

2.2.1 Facial Recognition

Con Facial Recognition, o riconoscimento facciale si intende l'identificazione di un utente mediante il proprio volto.

Una formulazione generale del problema potrebbe essere: dato un video o un'immagine di una scena, identificare o verificare la presenza di una o più persone nella scena usando un database di volti in memoria.[9]

Allo stato dell'arte esistono una moltitudine di algoritmi per il riconoscimento facciale, molti dei quali sfruttano tecniche e concetti assai complicati.

Gli algoritmi si possono dividere in due categorie:

- Basati sull'intensità dell'immagine (Eigenfaces,ICA,Kernel...)
- Basati sulla ricostruzione 3D del volto.

In entrambi i casi spesso l'algoritmo è implementato grazie alle reti neurali.[10]

Nel primo caso il riconoscimento si svolgerà generalmente in tre fasi:

- Face Detection
- Pre-processing (spesso eseguito insieme alla Face Detection)
- Face Recognition

2.2.2 Riconoscimento con Algoritmo Eigenfaces

L'algoritmo Eigenfaces è basato a sua volta dall'algoritmo PCA (Principal Component Analysis). È sicuramente il più semplice e immediato: il più adatto a questa tesi in quanto può essere implementato velocemente e anche se non offre buone prestazioni è sufficiente allo scopo.

In questo metodo le immagini sono rappresentate come vettori di lunghezza n pari al numero di pixel contenuti nell'immagine ottenuti concatenando le righe dell'immagine in un unica riga.

Per prima cosa è necessario individuare la presenza di un volto. Per far questo si può usare il metodo delle Haar-like Features [1].

Questo metodo usa sempre l'intensità dell'immagine per riconoscere se all'interno di essa c'è una determinata categoria di oggetti.

Per farlo l'algoritmo deve essere precedentemente addestrato tramite l'impostazione di soglie che in seguito saranno contenute in un “cascade file”.

In rete si trovano già vari cascade file che sono in grado di riconoscere vari oggetti e varie parti del corpo.

L'algoritmo restituisce un rettangolo che dovrebbe contenere l'oggetto o gli oggetti sulla base delle coordinate dell'immagine stessa.

Una volta rilevato il volto si può procedere al pre-processing.

Eigenfaces, essendo un algoritmo basato sull'intensità dell'immagine, ovvero sulla tonalità di grigio di ogni pixel, è molto sensibile a una serie di fattori che potrebbero creare problemi.

In primo luogo all'illuminazione in quanto le ombre presenti sul volto se poste in modo diverso rispetto alle foto campione vengono rilevate come una grossa differenza.

L'algoritmo è molto sensibile anche alla minima rotazione o traslazione dell'immagine in input rispetto a quelle campione.

Infine anche lo sfondo contribuisce agli errori.

È necessaria quindi una fase di pre-processing ben studiata per cercare di ridurre al minimo questi errori.

Innanzitutto è necessario usare foto in tonalità di grigio in quanto i colori non introducono nessun miglioramento ma contribuiscono solo ad appesantire la computazione.

La foto dovrà contenere solo ed esclusivamente il volto del soggetto cercando di escludere il più possibile lo sfondo, per esempio applicando una maschera ovale alla foto che mostri solamente il viso e annulli lo sfondo.

La foto andrà per quanto possibile ruotata e scalata in modo da mantenere verticale il viso, magari basandosi sulla posizione degli occhi. Infine per esaltare le caratteristiche salienti può essere eseguita un'equalizzazione dell'immagine che fa in modo che l'immagine copra tutta la gamma di grigi che vanno dal bianco puro al nero puro.

Infine si può passare all'algoritmo di riconoscimento vero e proprio.

L'algoritmo si sviluppa come segue.

Premesse: come già detto un'immagine in tonalità di grigio viene rappresentata come un vettore ottenuto concatenando tutte le righe dell'immagine.

Secondo la notazione il vettore si indicherà con **Γ** e sarà lungo **N**.

1. Per prima cosa si ottiene l'insieme **S** composto da **M** immagini di addestramento, ovvero il set di immagini con cui valutare la somiglianza rispetto all'immagine in input.
$$S = \{\Gamma_0, \Gamma_1, \Gamma_2, \ldots \Gamma_{M-1}\}$$
2. A questo punto si ottiene l'immagine "media" **Ψ**
$$\Psi = \frac{1}{M} \sum_{n=0}^{M-1} \Gamma_n$$
3. Si ottengono le immagini differenza tra quelle originali e la media
$$\phi_i = \Gamma_i - \Psi$$
4. In questa situazione si può trovare la matrice di covarianza nel modo seguente:
$$C = \frac{1}{M} \sum_{n=0}^{M-1} \underline{\phi_n} \underline{\phi_n}^T = A A^T$$
$$con\ A = [\underline{\phi_0}, \underline{\phi_1}, \ldots \underline{\phi_{M-1}}]$$

   Data questa matrice si possono trovare gli autovalori e gli autovettori in modo che soddisfino la seguente equazione.

$$\lambda_k = \frac{1}{M} \sum_{n=0}^{M-1} (\underline{u_k}^T \underline{\phi_n})^2 \; dove$$

λ_k *autovalore*

$\underline{u_k}$ *autovettore*

Gli autovettori formano una base ortonormale e un volto qualsiasi può essere proiettato su di essa, le coordinate così trovate indicheranno in che misura l'immagine data è composta dalle caratteristiche peculiari degli altri volti (Principal Component Analysis), queste caratteristiche sono contenute negli autovettori stessi che prendono il nome di EigenFaces o "auto-facce".

L'auto-faccia presente nella maggior quantità nell'immagine data sarà quella più somigliante e quindi il riconoscimento restituirà in uscita una label associata a quel particolare volto.

Purtroppo calcolare gli autovettori della matrice covarianza richiede una computazione pesante in quanto la matrice è NxN dove è N è il numero di pixel delle immagini, numero abbastanza alto.

Tuttavia essendoci solo M volti tra cui scegliere, le componenti con indice maggiore di M saranno sempre zero e quindi non vale la pena di calcolarle.

5. Partendo da questo presupposto si costruisce la matrice L

$$L = A^T A$$

Che sarà MxM, molto più piccola della matrice C

6. A questo punto si possono trovare gli M autovettori $\underline{v_l}$
7. Una volta trovati con il metodo del polinomio caratteristico o con altri metodi di analisi numerica, questi autovettori sono delle combinazioni lineari delle auto-facce che volevamo trovare in precedenza, e si possono trovare con la seguente formula

$$\underline{u_l} = \sum_{k=0}^{M-1} \underline{v_{lk}}\, \underline{\phi_k} \qquad l = 0,1,\ldots,M-1$$

Lo spazio formato dalle auto-facce prende il nome di face space, spazio dei volti.

Inoltre gli autovalori danno un'indicazione del peso che queste auto-facce hanno nel calcolo e quindi permettono di rendere ancora più preciso il riconoscimento del volto.

Sperimentalmente è stato verificato che per un buon riconoscimento del volto sono sufficienti una quarantina di componenti, in quanto le auto-facce calcolate successivamente presentano perlopiù rumore e non sono molto significative.

8. A questo punto un generico volto in ingresso, Γ, può essere proiettato nello spazio dei volti togliendo precedentemente il suo valore medio, ovvero il volto medio precedentemente calcolato.

$$\Omega = \{\omega_k\} \qquad \omega_k = \underline{u_k}(\Gamma - \Psi)$$

Da notare che questa è l'unica operazione che va eseguita in realtime, tutti i calcoli precedenti vengono eseguiti non appena vengono caricate le immagini campione.

9. Il vettore Ω contiene quindi le componenti dell'immagine proiettata sullo spazio dei volti, a questo punto si può scegliere il criterio di decisione per determinare il volto più somigliante, il più semplice è il criterio di minima distanza euclidea, ovvero si sceglie il volto campione che dista il meno possibile dall'immagine data.

La distanza euclidea può essere calcolata così

$$\epsilon_k^2 = \left|\Omega - \Omega_k\right|^2$$

dove Ω_k è il vettore rappresentativo dell'immagine campione k-esima.[11]

2.3 Ambienti Intelligenti

L'idea di ambiente intelligente si basa sul concetto di **Ubiquitous Computing**.

Il termine Ubiquitous Computing (UbiComp) fu coniato da Mark Weiser nel 1988, e indica un nuovo modello di interazione uomo-macchina.

Egli immaginava un futuro in cui l'ambiente circostante fosse letteralmente pervaso da dispositivi dotati di capacità di calcolo, come l'impianto di illuminazione o un elettrodomestico, e che questi si interfacciassero con l'utente in modo assolutamente semplice e discreto, magari senza neppure che egli se ne rendesse conto.

Da notare che l'UbiComp è l'esatto opposto rispetto alla realtà virtuale. Nella realtà virtuale l'utente si immerge nel mondo del computer, con l'UbiComp invece è il computer che viene immerso nel mondo fisico per interagire con esso.[12]

2.3.1 La rappresentazione della Conoscenza.

Con l'UbiComp si hanno dispositivi che lavorano in modo parallelo e pervasivo, tuttavia manca ancora un ingrediente perchè gli ambienti così realizzati siano davvero intelligenti.

Le informazioni attualmente sono rappresentate in rete come semplici testi. Tuttavia per lo scopo che ci prefiggiamo, questa rappresentazione è un po' carente in quanto i dispositivi non sono in grado di comprendere il significato di queste informazioni, ma al massimo le possono riportare all'utente che poi provvederà a capirle, ma questo è l'opposto di quanto vogliamo noi, così l'interazione con l'utente è tutt'altro che discreta.

Quello che serve è una rappresentazione della semantica delle informazioni ed è quello che si studia nell'ambito del **Web Semantico**.

Attualmente i motori di ricerca, si basano sulla ricerca di parole all'interno delle pagine sparse in rete, ma questo metodo di ricerca non è ottimale, richiede che l'utente perda parecchio

tempo a controllare tutti i risultati per trovare ciò che gli interessava, e inoltre il motore non sa veramente cosa sta cercando e potrebbe trovare pagine che non c'entrano nulla. Per esempio una ricerca di “Leonardo da Vinci” potrebbe restituire una pagina web con titolo “Questo sito parla di tutto tranne che di Leonardo da Vinci”.

Da anni viene studiato un modo più efficiente di rappresentare la conoscenza, esiste per esempio il linguaggio XML (eXtensibile Markup Language) che permette di rappresentare in un unico documento informazioni organizzate in una struttura ad albero grazie all'utilizzo di tag che possono essere creati dall'utente stesso, tuttavia questo linguaggio non è sufficiente in quanto la conoscenza non ha una struttura ad albero, e in quanto questi tag creati “ad hoc” dall'utente non possono essere compresi da qualsiasi dispositivo in quanto, tornando al problema iniziale, sono semplice testo.

La struttura della conoscenza è a grafo, ci possono essere oggetti o risorse che hanno delle proprietà che li legano ad altre risorse oppure a semplici valori alfanumerici.

Per esempio una matita può di essere colore giallo, in questo caso esistono due entità: la matita e il colore giallo, la matita possiede inoltre la proprietà di “avere un colore” che lega le due entità.

In un altro esempio si può pensare che quel monumento si chiami “Colosseo”, in questo caso l'entità è uno specifico monumento che ha la proprietà di avere un nome e questo nome è “Colosseo”.

In questo modo le informazioni sono rappresentate come un insieme di asserzioni, composte ognuna da un soggetto, un predicato e un oggetto.

Così la struttura che può assumere la conoscenza è molto più complicata ma è anche molto più espressiva.

Infine ogni entità, o nodo, viene memorizzato come Uri (Uniform Resource Identifier).

Per esempio il colore giallo potrebbe essere memorizzato come “http://it.wikipedia.org/wiki/Giallo“ quindi tutti i calcolatori che utilizzeranno questa particolare risorsa, sapranno che si tratta della stessa cosa e in particolare di quel colore.

Anche le proprietà stesse, i predicati, saranno memorizzati come Uri, la conoscenza infine sarà rappresentata in modo univoco e comprensibile da tutti.

Un Uri è composto principalmente da due parti: il namespace e il nome locale della risorsa. Questa suddivisione serve ad organizzare meglio le risorse in contesti specifici, tuttavia ogni qual volta si voglia scrivere un Uri nella sua versione completa, sarà sprecato un sacco di spazio in memoria in quanto sarà una stringa di una certa lunghezza. Esiste quindi la possibilità di dichiarare all'interno del contesto il namespace ed associarlo ad un prefisso, fatto questo un Uri può essere scritto come qName (qualified Name) ovvero il prefisso, breve, e il nome locale.

Per esempio il colore giallo come scritto sopra è identificato dall'Uri “http://it.wikipedia.org/wiki/Giallo“ che è formato dal namespace “http://it.wikipedia.org/wiki/” e dal nome locale “Giallo”.

Si potrebbe a questo punto associare il namespace al prefisso “wiki” dando la possibilità d'ora in poi di identificare il colore giallo mediante il qName “wiki:Giallo”.[13][14]

La rappresentazione dell'informazione descritta sopra è definita dallo standard RDF (Resource Description Framework) [15][16].

2.3.2 Ontologie e RDF Schema

In realtà quanto detto fin'ora non è ancora sufficiente a una rappresentazione univoca delle informazioni. Anche se una risorsa è identificata da un Uri, applicazioni diverse possono usare Uri diversi per identificare la stessa cosa, basti pensare che il colore giallo si può identificare come

“http://it.wikipedia.org/wiki/Giallo” ma anche come “http://en.wikipedia.org/wiki/Yellow”.

La soluzione a questo problema è fornita da un altro componente base del web semantico, collezioni di informazione chiamate **ontologie**.

In filosofia un ontologia è una teoria sulla natura dell'esistenza, e sui tipi di cose che esistono.

Nel web semantico l'ontologia definisce tutti i tipi di informazione che possono esistere in quell'ambito, dalla tassonomia, ovvero la suddivisioni in classi e sottoclassi, e delle regole di deduzione che permettono di ricavare nuova informazione automaticamente a partire da quella già esistente.

Questo è utile perchè sulla macchina che fornisce questo tipo di servizio, possono essere in esecuzione dei cosiddetti “reasoner”, programmi che analizzano appunto le informazioni e in base alle regole di deduzione ne creano di nuova.

Per esempio la proprietà di “trovarsi” da qualche parte è transitiva e questo è memorizzato nell'ontologia come regola di deduzione.

A questo punto se Tizio si trova a Cesena, e Cesena si trova in Italia, un reasoner può dedurre che Tizio si trova in Italia.

Il reasoner non è dotato di intelligenza artificiale, ma il sistema inizia a comportarsi in maniera molto più intelligente del solito grazie a questa organizzazione delle informazioni.

A questo punto, per risolvere il problema iniziale, basterà inserire nell'ontologia una relazione di equivalenza tra i due Uri che ora avranno lo stesso significato.

Per fornire i meccanismi per definire la tassonomia l'RDF viene esteso con l'RDF Schema (RDFS), una serie di predicati prestabiliti che permettono di creare la struttura in classi e la specializzazione di tutte le proprietà.

I più importanti sono

- Rdf:type indica l'appartenenza di quella risorsa a una classe
- Rdfs:subClassOf crea la relazione tra classe e sotto-classe

Per i meccanismi di deduzione l'RDF viene esteso con l'OWL (Ontology Web Language)[17] che analogamente all'RDFS specializza ulteriormente le proprietà.

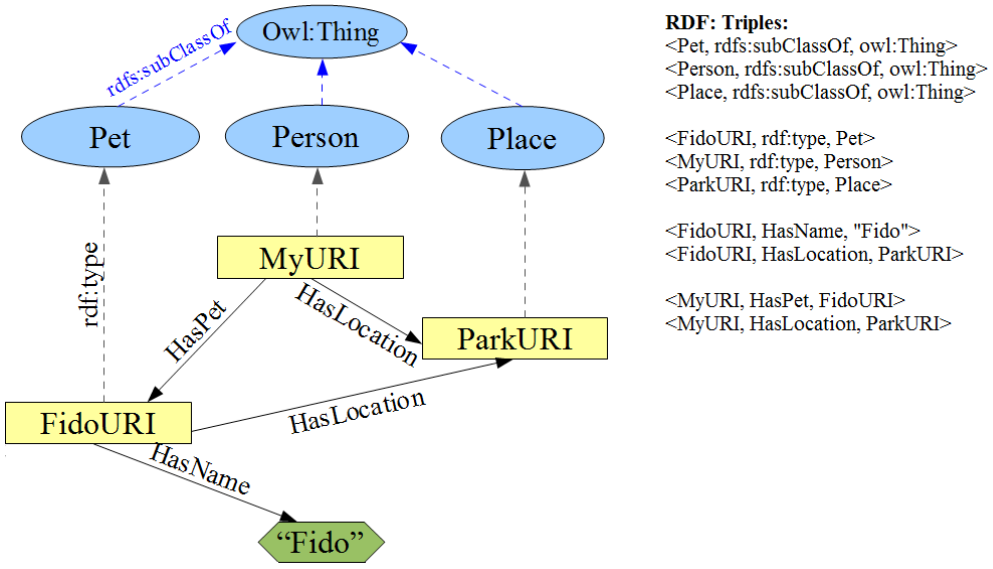


Figura 2.6: Esempio di informazioni organizzate a grafo

In Figura 2.6 è possibile osservare un esempio di informazione organizzata in un grafo secondo un'ontologia, con a lato le relative triple RDF.

L'Ontologia prevede una tassonomia formata dalla classe radice, owl:Thing, che in genere è sempre presente, e da tre sottoclassi che identificano gli animali domestici, le persone e i luoghi (in azzurro).

Quindi esiste un'istanza per ognuna delle classi identificata da un Uri che ovviamente non rappresenta una risorsa Web ma una fisica. Ogni entità ha le sue proprietà descritte dalle frecce nere, in sostanza la situazione mostrata rappresenta una persona col suo animale domestico, di nome Fido, al parco.

2.3.3 Ontologia per gli Ambienti Intelligenti

Gli ambienti intelligenti e le ontologie sono già da tempo oggetto di studio del centro di ricerca ARCES dell'Università di Bologna, è quindi una buona idea attenersi agli standard già definiti per la realizzazione di un progetto, in questo modo sarà compatibile con tutto ciò che è stato già sviluppato.

L'ontologia generale prevede una tassonomia formata da 5 classi principali: Data, Space, User, Device, Object (Figura 2.7).

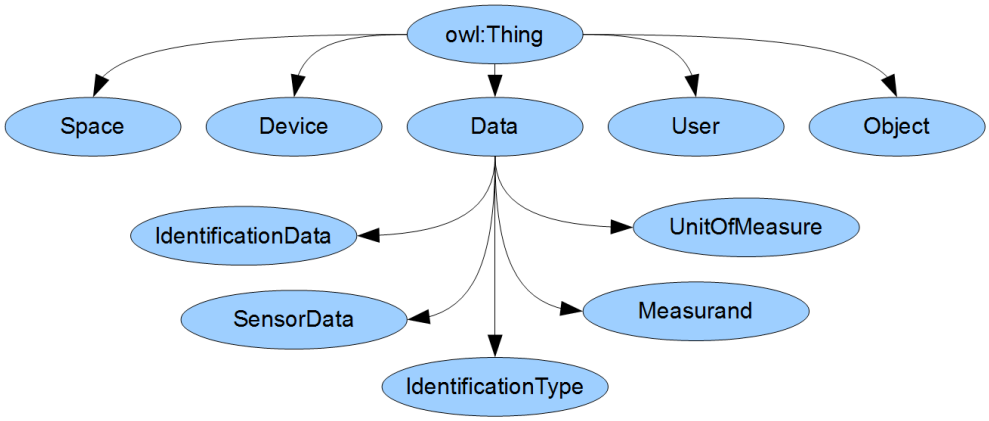


Figura 2.7: Tassonomia dell'ontologia di base

La classe Space conterrà tutte le informazioni sui luoghi presenti nell'ambiente intelligente, gli utenti che si muovono all'interno di esso saranno di classe User, i Device sono tutti i dispositivi dotati di capacità computazionale, come sensori smartphone o altro, gli Object sono gli oggetti inerti e infine tutte le informazioni sui dati saranno di tipo Data.

La classe Data si divide a sua volta nelle classi IdentificationData, che contiene i dati di identificazione di ogni entità, IdentificationType, che contiene i tipi di dati usati per l'identificazione che a loro volta sono contenuti nella classe Measurand, ovvero il tipo di grandezza fisica o dato (esempio Lunghezza, Potenza, QRCode...), SensorData, che contiene i dati prodotti da sensori, e infine UnitOfMeasure che contiene tutte le unità di misura per le grandezze fisiche.

Secondo questa ontologia possiamo immaginare una situazione in cui un dispositivo, DEV01, esegue un'identificazione dell'oggetto OBJ01 mediante QRCode.

Inizialmente OBJ01 avrà associato un IdentificationData con un certo valore (es. 0001) e IdentificationType QR-CODE.

Appena il dispositivo esegue la scansione, produrrà un SensorData che avrà lo stesso valore e Measurand sempre QR-CODE, come in Figura 2.8 [18].

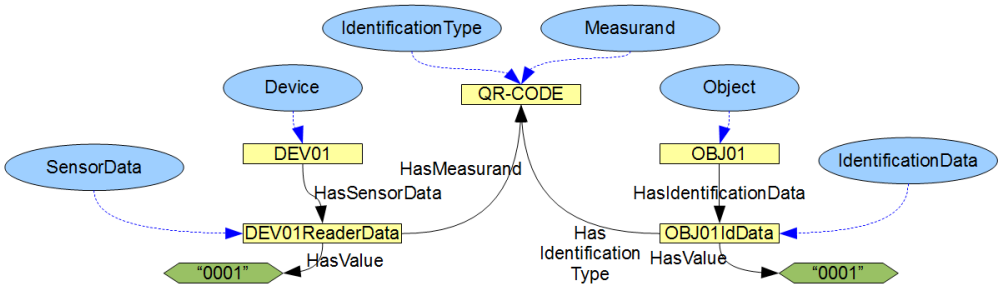


Figura 2.8: Esempio di Identificazione tramite QRCode secondo l'Ontologia generale

3 Strumenti Software

3.1 OpenCV per la Computer Vision

Figura 3.1: Logo di OpenCV

OpenCV (Open Source Computer Vision Library) è una delle più famose librerie Open Source per applicazioni di Computer Vision.

Inizialmente, nel 1999, fu sviluppata dalla Intel e attualmente è utilizzabile gratuitamente sotto licenza BSD.

La libreria è cross-platform (Android, iOS, Mac OS, Linux, Windows) e si concentra principalmente nel processing real-time. Se poi la libreria trova le Integrated Performance Primitives dell'intel installate sul sistema, le utilizza per ottimizzare i calcoli.

La libreria è sviluppata in C++ ma esistono altre versioni o altre librerie wrapper che ne permettono l'uso in linguaggi diversi, tra cui Python, Ruby, Java (JavaCV), C# e VB (Emgu CV)[19][20].

3.1.1 Funzionalità

Le funzionalità offerte dalla libreria si possono dividere in due categorie principali: Elaborazione delle Immagini e Machine Learning.

Per l'elaborazione delle immagini è possibile implementare gran parte degli algoritmi in modo semplice e veloce dato che molti di questi sono già realizzati all'interno delle API.

Per esempio il filtraggio o l'analisi di base delle immagini, l'analisi geometrica delle immagini (riconoscimento dei contorni), l'estrazione di caratteristiche (riconoscere figure geometriche o per esempio il punto di fuga prospettico), l'Inpaint, il riconoscimento della profondità con due telecamere, l'analisi del movimento come le gesture, riconoscimento di oggetti mediante gli Haar Classifier e riconoscimento facciale di base.

Il Machine Learning è implementato per fornire le funzionalità di base necessarie per alcuni algoritmi complessi come Decision Tree Learning o le Reti Neurali.

La libreria inoltre definisce anche i Datatype di base per l'utilizzo stesso, come le immagini, e tutti i costrutti matematici del caso come le matrici, inoltre vi sono primitive per l'acquisizione di immagini e video dal Filesystem o da periferiche come la webcam, e per la creazione di interfacce utente (GUI) di base.

3.1.2 Emgu CV

Per lo svolgimento della parte sperimentale è stata usata la libreria wrapper Emgu [21] in C#. Purtroppo la libreria non è compatibile ne con .Net Compact Framework (WinCE) ne per Windows Mobile Phone.

La libreria sarà usata quindi solo in ambito fisso su PC.

3.1.3 Utilizzo di Emgu CV per il riconoscimento facciale

Concentrandosi sull'obiettivo della tesi, e sapendo come si esegue il riconoscimento facciale (vedi Cap 1), illustrerò le primitive di interesse e come vengono utilizzate.

Le immagini sono implementate con la classe

```
Image<(Of <(<'TColor, TDepth>)>)>
```

dove Tcolor e Tdepth rappresentano rispettivamente il formato dei colori e la profondità in bit di ogni canale.

In genere Tdepth è sempre “byte” ovvero 8 bit, mentre Tcolor può essere “Bgr” nel caso si vogliano usare 3 canali dove il primo è il Blu, il secondo Verde e il terzo rosso, “Bgra” dove si aggiunge un quarto canale che rappresenta la trasparenza (Alpha Channel) oppure Gray dove si usa un solo canale e l'immagine sarà quindi in tonalità di grigio.

Con il costruttore si possono creare immagini da file specificando il percorso oppure a partire dalla classe Image che è quella standard usata da C#.

Un esempio di utilizzo può essere:

```
Image<Gray, byte> img = new
                    Image<Gray,byte>("C:\immagine.jpg");
```

Il costruttore esegue automaticamente la conversione tra i più importanti formati di immagini.

Questa classe fornisce già di per sé dei metodi per l'elaborazione di base dell'immagine, tra i più importanti per equalizzare l'immagine ovvero per fare in modo che il colore più chiaro sia il bianco puro e il colore più scuro il nero puro si può scrivere

```
Img._EqualizeHist();
```

Nel caso si vogliano acquisire fotogrammi dalla webcam, si crea un'istanza della classe Capture dalla quale a intervalli regolari si recupera il fotogramma corrente col metodo QueryFrame:

```
Capture webcam = new Capture(0);
for(;;){
  …
  Image<Bgr, byte> currFrame = webcam.QueryFrame();
  …
}
```

Nel creare l'oggetto "webcam" si specifica il numero 0, significa che si intende utilizzare il dispositivo di acquisizione con indice 0 ovvero, in genere, la webcam principale di sistema. Volendo si possono acquisire immagini da altri dispositivi.

All'inizio dell'algoritmo è necessario individuare il volto ed eseguire il pre-processing dell'immagine.

Per identificare un volto, o un oggetto qualsiasi, mediante le Haar Feature, si crea un'istanza della classe HaarCascade che contiene tutte le informazioni per l'identificazione.

```
HaarCascade HaarFace = new
                    HaarCascade("cascadeFilePath.xml");
```

A questo punto un metodo della classe Image esegue l'identificazione automaticamente passando alcuni parametri necessari al funzionamento dell'algoritmo.

```
MCvAvgComp[] faces = currFrame.DetectHaarCascade(HaarFace,
        1.05, 1, HAAR_DETECTION_TYPE.FIND_BIGGEST_OBJECT,
                                    new Size(100,100))[0];
```

Con questa riga di codice si esegue l'identificazione di un solo volto per volta che abbia dimensione minima 100 pixel per 100 pixel.

L'oggetto restituito è un vettore che contiene i dati di ogni volto rilevato, in questo caso uno solo. La posizione e dimensione del volto trovato sono rappresentati mediante la classe Rectangle e vi si può accedere in questo modo:

```
faces[0].rect
```

Dopo che l'immagine è stata pre-processata, verrà illustrato come nel capitolo successivo, si può eseguire il riconoscimento vero e proprio mediante EigenFaces.

Per prima cosa si recuperano le immagini di training e si mettono in un vettore di immagini rigorosamente in tonalità di grigio.

Quindi si crea un vettore di stringhe che conterrà una label per ogni immagine di training, l'algoritmo restituirà infine la label corrispondente all'immagine più somigliante a quella in ingresso.

Dopo aver specificato altri parametri riguardanti l'algoritmo si può creare l'oggetto EigenObjectRecognizer.

```
Image<Gray, byte>[] trainImg;
String[] labels;
MCvTermCriteria crit = new MCvTermCriteria(16,0.001);
// recupero delle immagini e delle label
EigenObjectRecognizer eor =
             EigenObjectRecognizer(trainImg,labels,ref
                                                  crit);
```

```
eor.EigenDistanceThreshold = 1250;
```

Con l'ultima riga di codice si dice al recognizer di non restituire nulla se l'immagine in ingresso non somiglia sufficientemente ad una delle immagini di training, altrimenti l'algoritmo restituirebbe la più somigliante anche se la somiglianza effettiva è oggettivamente minima.

Il valore è stato scelto empiricamente.

Fatto questo l'algoritmo si esegue con una riga di codice:

```
Image<Gray, byte> inputImage;
String outputLabel = eor.Recognize(inputImage);
```

3.2 ZXing per i codici a barre

Figura 3.2: Logo di ZXing

La decodifica di codici a barre si potrebbe tranquillamente implementare usando le funzionalità di base di OpenCV, tuttavia è presente in rete la libreria ZXing (Zebra Crossing) dedicata proprio a questo scopo.

Questa libreria è OpenSource e sviluppata inizialmente per Java, in particolare per l'utilizzo lato mobile ovvero su piattaforma Android.

Il linguaggio Java permette già di per se l'utilizzo della libreria su diverse piattaforme, in quanto il programma viene eseguito da una macchina virtuale, tuttavia esistono anche in questo caso varie versioni che permettono l'utilizzo in altri linguaggi, come C#, C++, iOS e ruby.

Gli sviluppatori hanno poi già creato un'applicazione di decodifica su Android completamente gratuita e scaricabile dal Market sotto il nome di barcodeScanner.[22]

3.2.1 Funzionalità

Come si è già intuito la libreria permette la lettura e decodifica di vari tipi di codici a barre lineari e a matrice, tra i più importanti UPC, EAN, QRCode e DataMatrix e oltre la decodifica è possibile anche la codifica in modo piuttosto semplice e sempre per gli stessi formati di codici a barre.

La struttura della libreria è piuttosto semplice, per ogni formato esiste una classe che implementa il decoder e una che implementa l'encoder, oltre ovviamente a tutte le altri classi usate nelle computazioni interne e che definiscono i Datatype.

Per esempio, nel caso di interesse dei QRCode, e prendendo in esame la versione della libreria in C#, la classe che si occupa della decodifica è QRCodeReader.

Per eseguire la lettura del codice bisogna istanziare una serie di oggetti che si occupano dell'elaborazione dell'immagine che potrebbe provenire per esempio dalla webcam, ottenuta attraverso le OpenCV già illustrate.

```
Capture c = new Capture(0);                    //OpenCV
Bitmap img = c.QueryFrame().ToBitmap();        //OpenCV
QRCodeReader r = new QRCodeReader();
RGBLuminanceSource ls = new RGBLuminanceSource(img,
                                               img.Width,
                                               img.Height);
HybridBinarizer bin = new HybridBinarizer(ls);
BinaryBitmap bufimg = new BinaryBitmap(bin);
```

Si possono specificare inoltre dei “suggerimenti” che in sostanza permettono all'algoritmo di saltare alcuni passaggi in modo da essere più veloce, per esempio se sappiamo già che i codici da leggere saranno solo di tipo QRCode possiamo scrivere il seguente codice.

```
Hashtable hints = new Hashtable();
Hints.Add(DecodeHintType.POSSIBILE_FORMATS,
                                    BarcodeFormat.QR_CODE);
```

Infine per elaborare l'immagine e trovare l'eventuale codice a barre:

```
Try{ //codice trovato
  Result res = r.decode(bufimg, hints); //res.Text contiene
                                               il risultato
}catch(Exception ex){
```

```
    //codice non trovato
}
```

L'encoding di un codice (QR) è un processo altrettanto semplice, questa volta l'encoder apparterrà alla classe QRCodeWriter.

```
QRCodeWriter qw = new QRCodeWriter();
Hashtable hints = new Hashtable();
Hints.Add(EncodeHintType.ERROR_CORRECTION,
                                ErrorCorrectionLevel.L);
Bitmap QRCodeImg =
  byteMatrixToImage(qw.encode("StringaDaCodificare",
                                BarcodeFormat.QR_CODE, 100,
                                      100, hints));
```

In questo caso verrà creata un'immagine di risoluzione 100x100 contenente il codice e che userà il livello di correzione degli errori specificato negli hints.

La funzione "byteMatrixToImage" in realtà non esiste all'interno della libreria, è stata creata ad hoc per riuscire a convertire il tipo di dati "ByteMatrix" nel formato standard per le immagini "Bitmap".

La funzione si implementa come segue:

```
private Image byteMatrixToImage(ByteMatrix o){
  Bitmap res = new Bitmap(o.Width,o.Height);
  for (int i = 0; i < o.Array[0].Length; i++){
        for (int j = 0; j < o.Array.Length; j++){
              if (o.Array[j][i] == 0)  res.SetPixel(i, j,
                                          Color.Black);
              else res.SetPixel(i, j, Color.White);
        }
  }
  return res;
}
```

3.2.2 ZXing per Android

Come già detto la libreria ZXing esiste anche in Java per Android, tuttavia si può sfruttare il sistema operativo e le applicazioni già installate per evitare di implementare da capo l'algoritmo di decodifica.

Per capirlo bisogna avere qualche nozione del funzionamento del kernel di Android e di come sono strutturate le applicazioni.

Ogni singola applicazione, o meglio ogni "schermata" di un'applicazione si rappresenta con la classe "Activity" che contiene, in sostanza, l'interfaccia utente dalla quale si possono lanciare altre computazioni o altre Activity.[23]

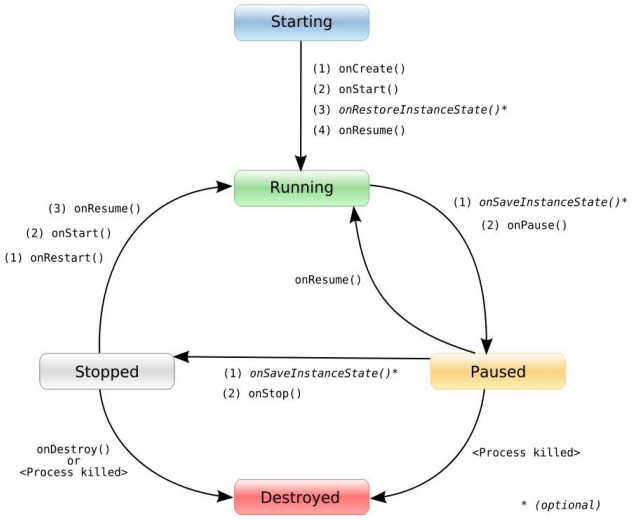


Figura 3.3: Ciclo di vita di una Activity in Android

In Android ci può essere una sola Activity in "primo piano", quando si lancia un'altra applicazione o si apre un'altra interfaccia utente, l'Activity corrente passa in secondo piano, o meglio va nello stato "Paused".

Un'applicazione che volesse lanciare una nuova activity deve eseguire una richiesta al Kernel che quindi gestirà la cosa e provvederà ad eseguirla.

La descrizione in stile Object Oriented di questo procedimento è che la prima Activity esprime l'intenzione di lanciarne un'altra, l'"intenzione" appartiene alla classe Intent.

In sostanza va definito prima l'Intent in cui si specifica il percorso dell'oggetto Activity che si vuole lanciare, secondo la suddivisione in package propria di Java, quindi la prima

Activity eseguirà una chiamata di sistema passandole in ingresso l'oggetto.

Tutto ciò è utile perchè come detto prima esiste già un'applicazione che legge i codici a barre creata all'interno dell'ambito del progetto ZXing: BarcodeScanner.

Si potrebbe evitare quindi di implementare di nuovo tutto il decoder e la parte di acquisizione delle immagini dalla fotocamera lanciando direttamente l'Activity che si occupa di tutto questo.

BarcodeScanner finita l'acquisizione restituirà poi all'Activity che l'ha lanciata una stringa contenente il contenuto del codice a barre letto.

Riassumendo la lettura del codice avviene in due righe di codice:

```
Private static final Intent SCAN_INTENT = new
        Intent("com.google.zxing.client.andoid.SCAN");
startActivityForResult(SCAN_INTENT, 0);
```

Il sistema operativo, una volta che la seconda Activity è stata terminata, ripristina la prima richiamando il suo metodo "onActivityResult" e passandogli tutti i valori di ritorno.

Il codice per ottenere finalmente il contenuto del QRCode è il seguente:

```
Protected void onActivityResult(int requestCode,
                                int resultCode,
                                Intent data){
  Super.onActivityResult(requestCode, resultCode, data);
  if(resultCode == Activity.RESULT_OK && requestCode == 0){
        String scanResult =
                    data.getStringExtra("SCAN_RESULT");
        //altro
  }
}
```

3.3 dotNetRDF

Lo scopo del progetto dotNetRDF è di creare una libreria Open Source usando il framework .Net di Microsoft che sia semplice da usare per lavorare con RDF [24].
Il fatto che sia stata creata con il framework .Net implica che si possa usare solo in ambiente Windows e in C#, inoltre la versione del framework è la 3.5.

Questa libreria è stata studiata in quanto potrebbe essere interessante a partire da un documento in linguaggio RDF/XML fare il parsing, ovvero l'analisi grammaticale del testo, e ricavare per esempio le triple descritte da quel documento.

3.3.1 Funzionalità e Utilizzo

In realtà le funzionalità offerte dalla libreria sono molto più varie di quanto descritto sopra.

Oltre al semplice parsing la libreria permette di gestire poi il contenuto informativo usando classi che rappresentano l'ontologia, permette di eseguire Query e di salvare le informazioni.

Supponendo che il documento RDF/XML si trovi sul Filesystem e che rispetti la sintassi propria del linguaggio [25] si procede a caricare il documento in un'istanza della classe Graph che implementa il grafo delle informazioni descritto nel capitolo 1, dal quale poi è possibile ricavare la collezione di triple che lo formano.

```
Graph g = new Graph();
RdfXmlParser p = new RdfXmlParser();
p.Load(g,"RDFData.xml");
foreach(Triple t in g.Triples){
  String sub = t.Subject.ToString();
  String pred = t.Predicate.ToString();
  String obj = t.Object.ToString();
  String nodeType = t.Object.NodeType.ToString();
                                        // Uri|Literal
}
```

Per l'operazione inversa, ovvero ottenere un documento in RDF/XML a partire dalle triple occorre creare e preparare un'istanza della classe Graph.

```
Graph g = new Graph();
g.NamespaceMap.AddNamespace("myns", new
              Uri("http://example.org/myNamespace#"));
…
Triple t = new Triple(g.CreateUriNode("myns:Me"),
        g.CreateUriNode("myns:HasPet"),
        g.CreateUriNode("myns:Fido"));
g.Assert(t);
…
RdfXmlWriter w = new RdfXmlWriter();
w.Save(g,"OutputFile.xml");
```

Da notare che nella prima parte è opportuno definire tutti namespace usati all'interno dell'ontologia.

La libreria supporta infatti i qName, che permettono di risparmiare una grande quantità di spazio nella scrittura del documento RDF/XML, tuttavia dato che i qName utilizzano un prefisso breve, questo va definito e associato al namespace intero come prima cosa.

Nell'esempio il namespace "http://example.org/myNamespace#" è stato definito e associato al prefisso "myns", in questo modo si possono scrivere qName come "myns:Fido".

3.4 Smart-M3

3.4.1 Introduzione

Smart-M3 è un progetto software Open Source che ha l'obiettivo di realizzare un'infrastruttura di condivisione delle informazioni tra entità e dispositivi per il Web Semantico, con lo scopo finale di realizzare gli ambienti intelligenti e il collegamento tra il mondo virtuale e quello reale.

Smart-M3 è tuttora sviluppato all'interno del progetto DIEM per gli ambienti intelligenti oltre che nell'ambito del progetto SOFIA, ma fu ideato inizialmente dalla Nokia come progetto proprietario.

L'obiettivo è di realizzare un ambiente intelligente secondo la filosofia descritta nel primo capitolo garantendo la massima interoperabilità tra i dispositivi che possono essere immersi nell'ambiente stesso.

Ciò significa che le informazioni devono essere rappresentate in modo comprensibile a tutti i calcolatori, una rappresentazione che incorpori anche la semantica dei dati stessi.

Inoltre un dispositivo per accedere al sistema dovrà soltanto conoscere il protocollo di comunicazione relativamente semplice che potenzialmente qualsiasi dispositivo, dotato di un minimo di connettività e di capacità computazionale, può implementare.

Ogni tipo di dispositivo di qualsiasi produttore.

Secondo le nozioni di sistemi distribuiti Smart-M3 è un sistema con architettura a “Spazio di Dati Condiviso”.

Una serie di programmi detti Agenti, o nella notazione propria di Smart-M3 “KP” (Knowledge Processor), realizzano la capacità computazionale del sistema stesso. Essi per scambiare informazioni tra di loro o verso il mondo esterno usano una sorta di repository, lo spazio di dati condiviso, che contiene tutte le informazioni di interesse, ogni agente quindi può accedervi e inserire informazione oppure ricevere una notifica in caso una certa informazione compaia all'interno di essa secondo un meccanismo di publish e subscribe.

Lo spazio di dati condiviso in linguaggio Smart-M3 prende il nome di SIB (Semantic Information Broker).

Il funzionamento del sistema sarà comunque approfondito in seguito.

I dati all'interno della SIB sono memorizzati in RDF che come detto in precedenza è il linguaggio ideale per

rappresentare ogni tipo di informazione in modo semantico.[26] [27]

3.4.2 Struttura e Filosofia di un'Applicazione

Com'è già stato anticipato i due componenti principali di Smart-M3 sono la SIB e i KP.

La SIB molto semplicemente è dove vengono memorizzate le triple RDF.

I KP invece accedono a queste informazioni per svolgere un compito ben preciso.

Da notare che la filosofia di Smart-M3 impone che ogni KP sia il più semplice possibile, per svolgere un compito complesso non ci vuole un KP complesso, ma tanti KP semplici e organizzati tra loro.

L'organizzazione tra i KP è dovuta al fatto che essi concordano sull'ontologia delle informazioni e sanno quali parti di queste informazioni sono di loro interesse. Tutto il resto verrà ignorato.

Infine il mezzo di comunicazione tra questi componenti è dato dal protocollo SSAP (Smart Space Access Protocol) che definisce i messaggi scambiati per ognuna delle operazioni che un KP può compiere nei confronti della SIB.

Queste operazioni primitive standard sono:

- **Join:** con la quale un KP accede alla SIB.
- **Leave**: con la quale un KP lascia la SIB.
- **Insert**: equivalente all'operazione generica “publish” con la quale il KP pubblica (atomicamente) delle informazioni nella SIB sotto forma di triple.
- **Remove**: con la quale vengono rimosse atomicamente delle triple
- **Update**: con la quale si modificano atomicamente delle triple, in realtà si tratta solo di una combinazione di remove e insert
- **Query**: con la quale un KP richiede delle informazioni contenute all'interno della SIB. Esistono due tipi di query:

- ***Query RDF***, con la quale si specifica un template di tripla e si ottengono tutte quelle che rispecchiano quel criterio, l'Uri “any” viene utilizzato in modo simile al carattere “*” nei sistemi operativi, ed esprime il concetto di “qualsiasi cosa”.
- ***Query Wilbur***, a partire da un nodo nel grafo e definito un percorso formato dai predicati delle triple si ottengono tutti i nodi collegati al nodo di partenza da quel percorso.

- **Subscribe**: un KP si può registrare a una certa collezione di triple, quando una di queste viene creata o modificata riceve una notifica con il suo nuovo valore in un'ottica Event Based.
- **Unsubscribe**: per annullare la sottoscrizione.[28]

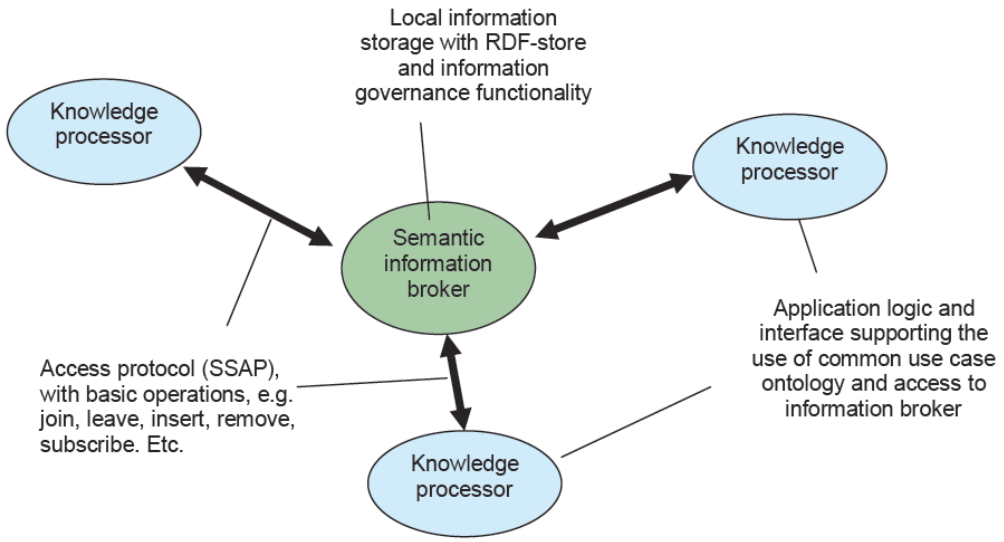


Figura 3.4: Visuale del Livello Informativo di Smart-M3

Dato il protocollo si possono suddividere i KP in tre classi in base alle operazioni che compiono.

- **Producer**: quando un KP produce nuova informazione e quindi esegue prevalentemente delle insert (o update)
- **Consumer**: quando il KP utilizza le informazioni nella SIB per mostrarle all'utente oppure per azionare degli attuatori, esegue prevalentemente delle subscribe.

- **Aggregator**: quando invece il KP a partire da informazioni già presenti nella SIB ne crea di nuove eseguendo dei ragionamenti, esegue sia insert che subscribe.

È naturale che ogni tipo di KP in fase di inizializzazione può eseguire delle query per venire a conoscenza per esempio dello stato dell'ambiente.

È consigliabile fare in modo che ogni KP faccia parte di una sola di queste classi e che si concentri solamente sul suo scopo in modo da essere il più semplice possibile.

Se rispettato questo requisito un'applicazione può diventare estremamente modulare, permettendo l'aggiunta o la rimozione di feature dinamicamente senza creare problemi al resto dell'applicazione.

Per capire questa grande potenzialità si può immaginare una situazione in cui in una stanza esistono un sensore di temperatura e un piccolo display che visualizza in tempo reale la temperatura.

In questo sistema il sensore di temperatura conterrà un KP di tipo producer che invia regolarmente alla SIB le informazioni sulla temperatura, il display invece conterrà un KP consumer che ogni volta che il sensore manda un nuovo dato si limiterà a visualizzarlo.

Il sistema descritto è estremamente semplice e modulare, infatti nel caso si volesse aggiungere una nuova funzionalità, per esempio l'invio di un SMS di allarme nel caso la temperatura scendesse sotto una certa soglia per prevenire il congelamento delle tubazioni, basterà aggiungere un terzo KP consumer sottoscritto a sua volta alle informazioni sulla temperatura che in caso di superamento della soglia inoltrerà il messaggio di allarme grazie ad hardware apposito.

Un upgrade di un sistema proprietario e progettato solo per la visualizzazione della temperatura sarebbe stato molto più complicato se non impossibile senza stravolgere completamente l'architettura di sistema.

3.4.3 I KP e le API

Dal sito ufficiale del progetto Smart-M3 [29] è possibile scaricare i sorgenti dell'ultima versione della SIB, attualmente la 0.9.5 beta.

La SIB al momento esiste solo per piattaforma linux ed è composta da vari moduli e pacchetti alcuni dei quali sono direttamente scaricabili dalle repository registrate nelle più comuni utility di installazione presenti, per esempio, in Ubuntu.

Una volta installata seguendo le istruzioni scaricabili dal sito si può lanciare eseguendo prima il demone “sibd” dal terminale, quindi attivando il servizio per l'interfacciamento col protocollo TCP “sib-tcp”. Questi programmi devono rimanere in esecuzione.

Salvo diverse configurazioni sarà aperto un nuovo spazio delle informazioni di nome “X” e sulla porta 10010. L'indirizzo IP ovviamente dipende dal server stesso.

Dal sito si possono scaricare le API lato KP per C++ e Python, tuttavia esistono anche gli equivalenti in C#[30], Java [31] e PHP [32].

Per lo svolgimento della tesi sono stati prese in esame le API per C# e Java, quest'ultimo utilizzato poi anche in ambiente Android.

È necessaria una piccola precisazione, ovvero che le API per C#, rilasciate da poco, sono più evolute di quelle in Java, infatti nella versione precedente e nella versione Java tutti i metodi di base SSAP ritornavano una stringa contenente il codice XML proveniente dalla SIB con le informazioni sulla riuscita dell'operazione o sui dati delle query, quindi un oggetto parser avrebbe dovuto processare questa stringa per ricavare le vere informazioni. Nell'ultima versione il parsing è incorporato nei metodi che ritorneranno un booleano oppure una collezione di triple per le query.

Core e Parser

L'oggetto principale per l'utilizzo di Smart-M3 è un'istanza della classe KPICore, al momento della creazione vanno

specificate le coordinate in rete della SIB, indirizzo IP e Porta in genere 10010, e il nome dello spazio intelligente, che come già detto salvo diversa configurazione sarà “X”.

```
KPICore.KPICore core = new
                    KPICore.KPICore(“10.0.0.1”,10010,”X”);
```

L'equivalente in Java sarà

```
KPICore core = new KPICore(“10.0.0.1”,10010,”X”);
```

In Java inoltre andrà creato l'oggetto parser in questo modo

```
SSAP_XMLTools parser = new SSAP_XMLTools(
              “00000000-0000-0000-0000-DEAD0000BEEF”,
              ”X”,”http://ww.nokia.com/NRC/M3/sib#any”);
```

Insert e Update, Join e Leave

In entrambi i linguaggi le triple vengono rappresentate con un vettore di stringhe lungo 4.

Gli elementi del vettore conterranno quindi in ordine soggetto, predicato, oggetto, e tipo dell'oggetto ovvero Uri oppure literal.

Quando si vuole fare un'insert un'update il metodo può richiedere i 4 campi della tripla separatamente nel caso in cui se ne voglia aggiungere solo una, in caso di più triple si passa in ingresso un “grafo” ovvero una collezione di triple che viene inserito atomicamente risparmiando notevole tempo sull'esecuzione perchè ci si risparmia l'overhead dato dall'instaurazione della comunicazione con la SIB.

Si ricorda che un Uri per essere valido deve contenere anche il namespace, purtroppo la SIB attualmente non supporta i qNames quindi l'Uri sarà da inserire per intero.

Un esempio di inserimento di grafo potrebbe essere per il C#

```
ArrayList graph = new ArrayList();
graph.Add(new String[]{“sogg1”,”pred1”,”ogg1”,”Uri”});
graph.Add(new String[]{“sogg2”,”pred2”,”ogg2”,”Uri”});
…
```

```
if(core.join()){
  If(!core.insert(graph)) MessageBox.Show("Errore nella
                                                    Insert");
  If(!core.leave()) MessageBox.Show("Errore nella Leave");
}
Else
{
  MessageBox.Show("Errore nel collegamento con la SIB");
}
```

Si può osservare che prima di eseguire delle operazioni sulla SIB è necessario eseguire la join e successivamente la leave.

In Java il codice analogo è

```
Vector<Vector<String>> graph = new
                                Vector<Vector<String>>();
graph.add(parser.newTriple("sogg1","pred1","ogg1",
                                          "Uri","Uri"));
graph.add(parser.newTriple("sogg2","pred2","ogg2",
                                          "Uri","Uri"));
…
if(parser.isJoinConfirmed(core.join()){
  if(!parser.isInsertConfirmed(core.insert(graph))
                                          debug("error");
  if(!parser.isLeaveConfirmed(core.leave()) debug("error");
}else{
  debug("errore nella connessione");
}
```

Il metodo newTriple del parser aiuta a creare il vettore di stringhe rappresentativo della tripla, si può notare che ci sono 5 argomenti invece di 4. L'argomento in più è il tipo del soggetto, ma questa informazione è inutile in quanto dovrà sempre essere un Uri, infatti nella versione in C# non compare.

Query RDF

Il metodo per le query RDF restituisce una collezione o un vettore di triple.

Nelle query RDF per definire i termini di ricerca si passa in ingresso al metodo un “template” di tripla, ovvero usando l'Uri speciale “any” si descrive una categoria ampia di triple che la SIB successivamente restituirà.

Ad esempio specificando il template (“any”,”rdf:type”,”User”) si ottengono delle triple i cui soggetti sono tutti gli utenti del sistema.

In C# il codice per questa ricerca è il seguente

```
ArrayList res =
        core.queryRDF(“any”,”rdf:type”,namespace+”User”);
foreach(String[] t in res){
  Console.WriteLine(“trovata tripla:
                    (“+t[0]+”-”+t[1]+”-”t[2]”(“+t[3]+”))”);
        // => trovata tripla: (sogg-pred-ogg(Uri))
}
```

In Java:

```
Vector<Vector<String>> res = parser.getQueryTriple(
                          core.queryRDF(null, ”rdf:type”,
                                namespace+”User”,”Uri”));
for(int i=0;i<res.size();i++){
  Vector<String> t = res.get(i);
  System.out.println(“trovata tripla: (“
                    +parser.triple_getSubject(t)+”-”
                    +parser.triple_getPredicate(t)+”-”
                    +parser.triple_getObject(t)+”(“
                    +parser.triple_getObjectType(t)+”))”);
}
```

In Java l'Uri “any” viene sostituito con l'oggetto null.

Query Wilbur

Le query Wilbur, a differenza delle query RDF, restituiscono in uscita una collezione di nodi del grafo piuttosto che di triple.

Un nodo si può definire come una coppia di stringhe, nel caso in cui il nodo sia un'istanza allora la prima stringa sarà il

suo Uri e la seconda sarà “Uri”, nel caso in cui sia un valore letterale la prima sarà il valore stesso e la seconda “literal”.

Una query Wilbur ha lo scopo di trovare tutti i nodi nel grafo che a partire da un nodo di partenza sono collegati ad esso tramite un percorso specificato.

Per prima cosa va definito il nodo di partenza. Il nodo consiste semplicemente in un vettore di stringhe lungo due e in C #deve essere creato “a mano”:

```
String[] StartNode =
                new String[]{ “IstUri|Value”, “Uri|literal”};
```

In Java invece non ce n'è bisogno in quanto le primitive richiedono come argomenti l'Uri e il tipo del nodo di partenza.

Una volta definito il nodo di partenza, va definito il percorso che va dal nodo di partenza fino ai nodi che vogliamo ottenere alla fine della query.

Il percorso viene definito secondo la sintassi descritta nel documento [33], il manuale di Piglet, un'infrastruttura software all'interno della SIB che si occupa di gestire il grafo delle informazioni.

Il percorso si esprime come un “pattern” che deve rispettare questa struttura:

```
Pattern ::= '[' operator { ',' pattern }* ']' | atom
```

Gli operatori di interesse in questo contesto sono “seq” che serve per esprimere una sequenza di elementi, e “inv” che esprime il concetto di inversione dell'elemento al quale si riferisce.

Le proprietà, ovvero i predicati delle triple presenti nel grafo, sono degli atom.

Il percorso è di fatto una sequenza di proprietà alcune delle quali possono essere percorse in senso inverso, partendo da questi presupposti si può facilmente costruire una stringa che esprime un percorso qualsiasi all'interno del grafo. La stringa sarà del tipo:

```
String path = "['seq' , 'prop1' , ['inv' , 'prop2'] ,
                          ['inv' , 'prop3'] , 'prop4']";
```

Dove prop1 e prop4 sono proprietà da percorrersi nel senso giusto, mentre prop2 e prop3 nel senso inverso. La situazione è rappresentata in Figura 3.5: una query Wilbur per la quale venisse specificato il percorso sopra indicato restituirebbe in output i due nodi in arancione.

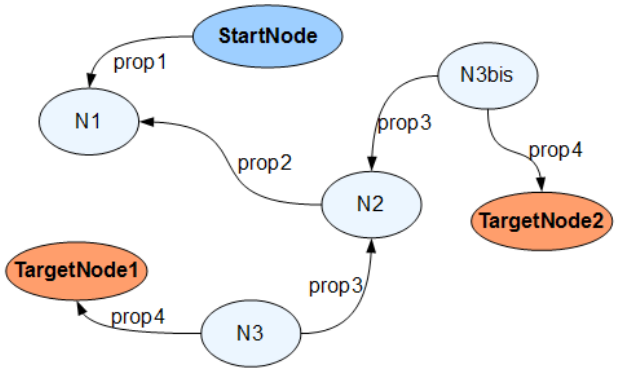


Figura 3.5: Esempio di definizione di percorso nel grafo

Detto questo per eseguire la query vera e propria ed ottenere i risultati si utilizzano gli appositi metodi: in C#,

```
ArrayList nodes = core.queryWQL_values(StartNode,path);
```

In Java,

```
Vector<Vector<String>> nodes =
  parser.getWQLResultNode(core.queryWQL_VALUES( "istUri|
                          value","Uri|literal",path));
```

Per accedere ai nodi risultato si procede analogamente alle queryRDF, in C#,

```
foreach(String[] node in nodes){
  Console.WriteLine("trovato nodo: ("+node[0]+" :
                                          "+node[1]+")");
        //trovato nodo: (IstUri : Uri)
        //trovato nodo: (value : literal)
}
```

In Java,

```
for(int i=0;i<nodes.size();i++){
  Vector<String> node = nodes.get(i);
  System.out.println(" trovato nuovo nodo: (" +
```

```
                    Parser.node_getValue(node) + " : " +
                    Parser.node_getType(node) + " )");
            //trovato nodo: (IstUri : Uri)
            //trovato nodo: (value : literal)
    }
```

Le query Wilbur sono molto utili per verificare, per esempio, a che classe appartiene un'istanza od operazioni simili. Nelle ultime versioni delle API sono stati introdotti infatti dei metodi che realizzano automaticamente queste verifiche.

- **queryWQL_related(String[] StartNode, String[] EndNode, String path)** verifica se lo StartNode e l'EndNode sono collegati dal percorso specificato
- **queryWQL_nodeTypes(String IstanceUri)** ritorna la classe (o le classi) di appartenenza.
- **queryWQL_isType(String IstanceUri, String ClassUri)** verifica se l'istanza appartiene alla classe specificata.
- **queryWQL_isSubType(String IstanceUri, String ClassUri)**, analoga alla precedente, ma considera anche l'albero delle classi e considera l'istanza appartenente a una classe padre della classe di appartenenza diretta.

In Java i metodi sono analoghi anche se cambiano i data type usati.

Subscribe RDF

L'utilizzo della subscribe è più complesso, analogamente alle query bisogna passare in ingresso i template delle triple alle quali ci si vuole sottoscrivere, inoltre si dovrà definire una classe che implementi l'interfaccia “iKPIC_subscribeHandler” che offrirà in particolare il metodo “kpic_SIBEventHandler” che verrà richiamato nel caso in cui la SIB invii una notifica al KP sulla modifica di una o più triple.

In C# la classe handler può essere definita in questo modo

```
Public class mySubscribeHandler: iKPIC_subscribeHandler
{
  Public void kpic_SIBEventHandler(ArrayList newResults,
              ArrayList obsoleteResults, string subID)
  {
        …
        //gestisco l'evento
        …
  }
}
```

Gli ArrayList newResult e obsoleteResults contengono una lista delle triple rispettivamente dopo essere state modificate e prima di essere state modificate e possono essere navigate similmente alle query RDF con il costrutto “foreach”.

In Java cambia lievemente il funzionamento del metodo in quanto prende in ingresso la stringa XML proveniente dalla SIB, il parser si occuperà poi di recuperare le singole triple, come al solito, similmente alle query RDF.

```
Public class mySubscribeHandler implements
                                  iKPIC_subscribeHandler{
  Public SSAP_XMLTools parser=null;

  Public void kpic_SIBEventHandler(String xml){
        Vector<Vector<String>> triples =
                    parser.getNewResultEventTriple(xml);
        …
        //gestisco l'evento
        …
  }
}
```

Una volta creata questa classe, all'interno del programma principale si usa uno dei metodi dell'oggetto KPICore passandogli in ingresso il template delle triple e la classe stessa.

Il metodo subscribe creerà un nuovo Thread che rimarrà in ascolto in attesa di eventi SIB, quindi al momento opportuno lancerà il metodo della classe appena creata in modo concorrente.

Nell'esempio si vuole ricevere una notifica nel momento in cui un'entità qualsiasi entra nella stanza “Stanza01”.

In C#:

```
MySubscribeHandler handler = new mySubscribeHandler();
If(!core.subscribeRDF(“any”, ”HasLocation”, ”Stanza01”,
        ”Uri”, handler))  MessageBox.Show(“errore nella
                                              Subscribe”);
```

In Java:

```
MySubscribeHandler handler = new mySubscribeHandler();
Handler.parser=parser;
Core.setEventHandler(handler);
If(!Parser.isSubscriptionConfirmed(core.subscribeRDF(null,
                    “HasLocation”, “Stanza01”, “Uri”))
                          debug(“Errore nella Subscribe”);
```

Subscribe Wilbur

Le subscribe di tipo Wilbur funzionano nello stesso modo di quelle RDF.

Il metodo per iniziare l'operazione è per C#

```
Core.subscribeWQL_VALUE(String[] StartNode, String path,
                                        myHandler handler);
```

Per Java

```
Core.subscribeWQL_VALUES(String StartNodeUri, String
                            StartNodeType, String path);
```

L'unica differenza è che bisogna passare al metodo i parametri tipici delle query Wilbur, ovvero il nodo di partenza e il percorso nel grafo.

L'oggetto handler da utilizzare è lo stesso descritto nelle subscribe RDF, l'unica differenza è che nel metodo

kpic_SIBEventHandler i parametri in ingresso sono da interpretarsi come nodi e non come triple.

In Java esistono altri metodi aggiunti analoghi a quelli descritti nel paragrafo sulle Query Wilbur per rendere più immediato il riconoscimento del tipo di appartenenza o se due nodi sono collegati da un percorso.

4 Esempio Applicativo

4.1 Scopo della Sperimentazione

Uno dei principali obiettivi di Smart-M3, come è già stato detto, è di offrire la massima interoperabilità ai sistemi che lo usano.

In questo caso si vogliono implementare funzionalità di identificazione, attraverso le tecniche già illustrate, per applicazioni general purpose.

All'interno del gruppo di ricerca Arces dell'Università di Bologna sono già state studiate tecniche di identificazione in ambiente Smart-M3 mediante l'uso di tag RFID, l'obiettivo è quello di ottenere lo stesso risultato mediante i QRCode o mediante il riconoscimento del volto nel caso di utenti.

L'implementazione sarà indipendente dallo scopo finale dell'applicazione, ovvero sarà oggetto di studio come inserire nell'ambiente l'informazione sull'avvenuta lettura di un codice o di un volto, senza aggiungere altro.

Sarà compito di altre parti del sistema, alcune implementate ex-novo altre già esistenti, ricavare informazioni di interesse o compiere azioni a partire dall'evento di riconoscimento.

In questo modo si evidenzia il forte carattere modulare che ha un'applicazione Smart-M3.

Lo scopo finale sarà di dimostrare l'ampio grado di interoperabilità garantito da Smart-M3 mostrando come si possono usare tecnologie diverse e apparentemente incompatibili per ricavare informazioni che, a loro volta, possono essere usate da qualsiasi altra tecnologia.

4.2 Realizzazione del Sistema

4.2.1 Concept

Per riuscire a sperimentare con successo la parte software oggetto di questa tesi è stato comunque necessario pensare a un possibile scenario in cui si può utilizzare.

Come punto di partenza si vuole realizzare un sistema di localizzazione basato sull'identificazione di oggetti e utenti con le tecniche di Computer Vision mostrate nel capitolo 2, dove l'identificazione della stanza avviene mediante QR-Code.

Il tutto si basa sul fatto che ogni spazio dell'ambiente intelligente è identificato da un QRCode. Un dispositivo che entra in uno spazio deve leggere il codice mediante l'apposito KP. A questo punto il sistema può dedurre che il suddetto dispositivo si trova in quel particolare spazio. Se per esempio si sapesse che un certo utente in quel momento ha con sé quel dispositivo allora il sistema potrebbe dedurre che anche l'utente si trova in quello spazio.

Similmente se il dispositivo identifica un utente con il riconoscimento facciale, e il dispositivo si trova in un certo spazio, anche l'utente si trova in quello spazio.

In sostanza il concetto di base è che la proprietà di “Trovarsi in un luogo” è transitiva e che la proprietà di “possedere un dispositivo” implica “trovarsi nello stesso luogo del dispositivo”.

4.2.2 Descrizione dell'Applicazione

Il formalismo con il quale si andrà a descrivere l'applicazione è enunciato nel documento [34] della bibliografia.

Lo scenario è già stato descritto, per gli esperimenti è stato allestito un ambiente intelligente composto da 2 stanze, 2 utenti, un dispositivo mobile Android e un PC fisso. Ad ogni stanza è stato associato un QRCode, come anche al dispositivo fisso, e ad ogni utente sono state associate alcune immagini del loro volto.

SCENARIO DESCRIPTION			
F_ID	**FIELD_NAME**	**Value**	**Notes**
1_1	**Short Name**	FandQRID	*Insert an acronym to be used as a reference in other templates (Max 8 Char)*
1_2	**Title**	QRCode and Face Recognition based Localization	*A title for the scenario*
1_3	**Short Narrative Description**	Devices, working in the Smart Space, can recognize some QRCode or User's faces. They use this informations to deduce users and object location. The scenario includes the following entities: • 2 Rooms • 2 Users • A mobile phone equipped with a camera • A Laptop equipped with a webcam.	*Use cases are short stories of envisioned real life situations that describe a specific user group in a specific contextual setting interacting with specific devices to fulfill specific tasks and intentions*
1_4	**Reference Person**	Luca Feltrin (luca.feltrin@studio.unibo.it)	*Insert name and contact of the team leader and team_ID*
1_5	**Date**	17/08/11	*Insert date of scenario creation*

USERS			
F_ID	**USER_ID**	**Type**	**Role**
2_1	**U1**	General User	People moving around

ARTIFACTS			
F_ID	**A_ID**	**Type**	**Description**
3_1	**A1**	Nomadic device	Smartphone with Camera
3_2	**A2**	Room Main Device	Laptop located in a Room with WebCam
3_3	**A3**	Server	Server running the SIB and Apache

Tabella 4.1: Descrizione dello Scenario Applicativo

Come si evince dalla Tabella 4.1 il sistema necessita di un Server che contenga ovviamente la SIB, ma anche un server Web che possa contenere il database delle foto degli utenti. In generale questo server può essere ovunque nel mondo, esistono infatti tanti siti web che forniscono già il servizio di Hosting delle immagini, per l'applicazione le immagini sono state caricate sui server di imageshack.[35]

La SIB durante la fase di testing è stata installata su un PC all'interno della rete locale. Successivamente è stata usata la

SIB pubblica del gruppo Arces all'indirizzo “mml.arces.unibo.it”.

In generale ci possono essere vari dispositivi sia di tipo A1 che A2 e vari utenti. Anche qui per semplicità si è scelto di eseguire l'esperimento con un numero ridotto di entità.

La configurazione dell'applicazione è descritta nella Figura 4.1.

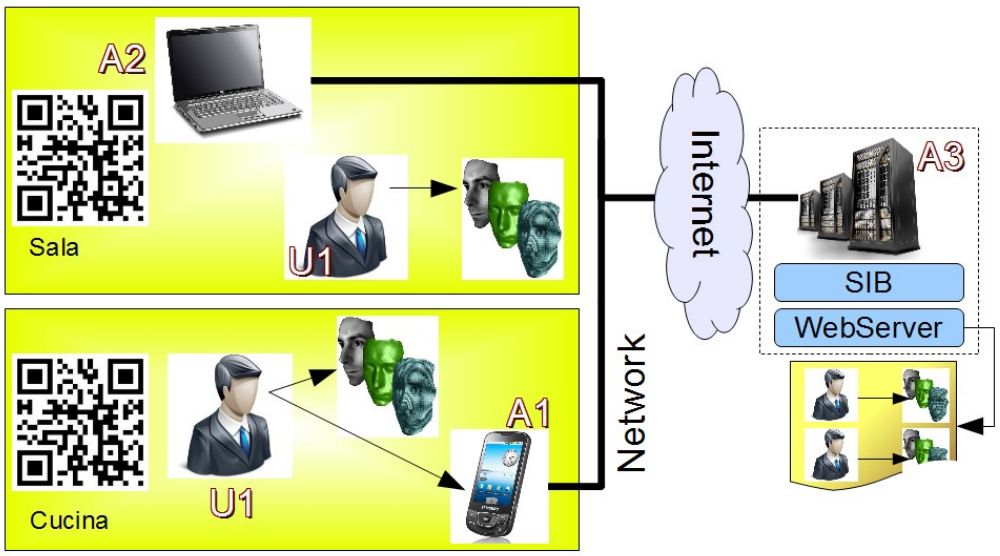


Figura 4.1: Layout dell'applicazione

A questo punto è necessario esaminare il problema più a fondo per determinare i singoli componenti software, ovvero i KP, che andranno a risolverlo.

Nell'ottica di massimizzare la modularità dell'applicazione si può suddividere il problema in tre parti.

- Nella prima parte avviene l'acquisizione delle informazioni, ovvero l'inserimento nella SIB dell'evento di acquisizione di un QRCode o di un volto.

  Dato che questa parte deve essere general purpose, si è pensato che i KP di tipo producer, inseriranno semplicemente nella SIB una tripla che notificherà il fatto che quel dispositivo ha rilevato un nuovo dato, di tipo “Volto” o “QRCode” similmente a quanto farebbe un sensore.

Un KP di nome KP_FaceAdapter si occuperà del riconoscimento dei volti, mentre KP_QRCodeAdapter si occuperà della lettura dei codici a barre, quest'ultimo esisterà in due versioni, una per Android e una per Windows.

- Successivamente, a partire dal rilevamento da parte dei dispositivi, si potranno ricavare le informazioni circa la localizzazione vera e propria.
  KP_LocationAggregator si occuperà di aggiungere la proprietà “HasLocation” agli oggetti identificati dai due KP producer. In particolare nel caso l'oggetto rilevato sia uno spazio allora asserirà che il dispositivo responsabile dell'identificazione si trova in quel luogo, nel caso sia un altro tipo di oggetto asserirà che quell'oggetto ha la stessa locazione, se disponibile, del dispositivo.
  In particolare è stato deciso che se sia il dispositivo che l'oggetto identificato avessero già una precedente locazione la più obsoleta venisse aggiornata in quella più recente delle due. Per memorizzare quando un oggetto assume una nuova locazione è stata creata nell'ontologia una nuova proprietà “HasLocationTimestamp”.
  Data l'esistenza nell'ontologia della proprietà “HasDevice” si può inserire un terzo Aggregator, KP_PossessionAggregator, che a partire dalle informazioni di localizzazione appena create, se un utente viene localizzato in una stanza e possiede un dispositivo anche il dispositivo viene localizzato in quella stanza e vice versa.
- Una volta che le informazioni sulla localizzazione sono presenti, altri KP possono sfruttarle per qualsiasi scopo.
  A questo scopo è stato creato un KP di tipo consumer, KP_LocationMonitor, che visualizzerà con un'interfaccia grafica una mappa degli ambienti e le entità contenute in essi.

Riassumendo quanto detto lo schema finale dell'applicazione sarà quello descritto in Figura 4.2.

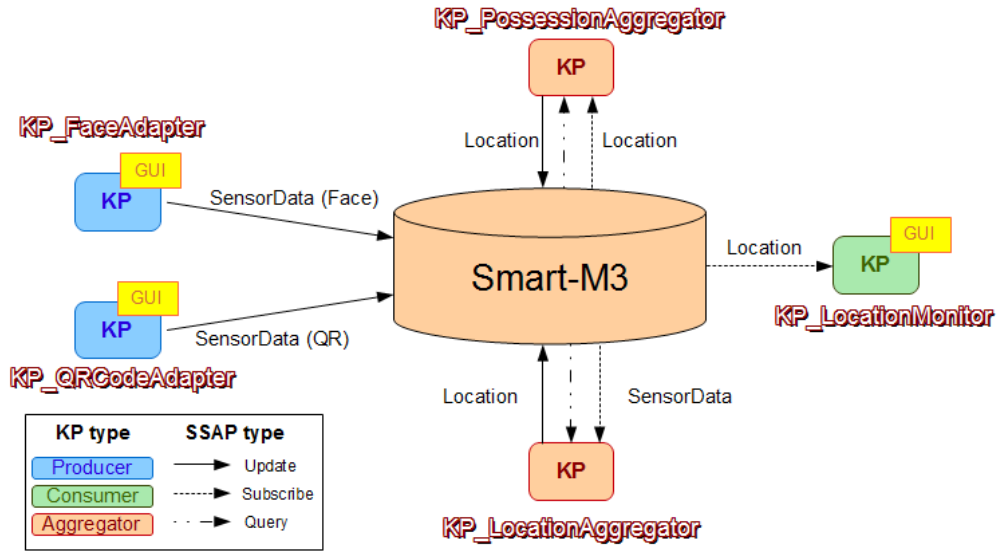


Figura 4.2: Architettura Software

Il flusso delle informazioni a tempo di esecuzione descritto in Figura 4.3 mostra che molto semplicemente i due producer generano dati di tipo SensorData, questi vengono elaborati dal LocationAggregator che genera le informazioni sulla locazione che vengono mostrate dal LocationMonitor ed elaborate ulteriormente da PossessionAggregator.

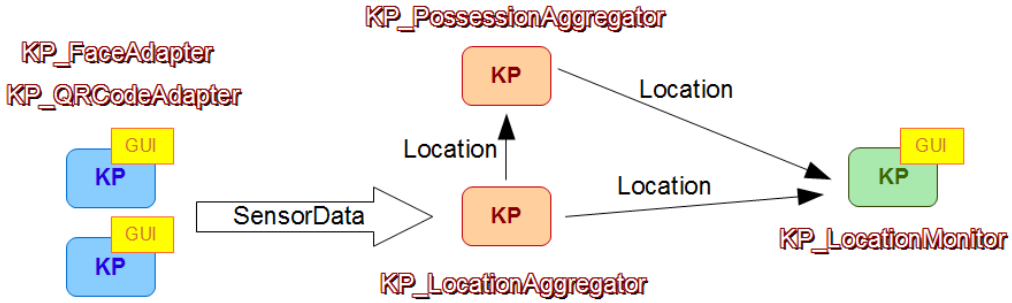


Figura 4.3: Flusso delle Informazioni

Scendendo più nella parte tecnica, va deciso dove eseguire i vari KP, su quale sistema operativo e in quale linguaggio. L'artefatto “A3”, ovvero il calcolatore sul quale è in esecuzione la SIB può essere il luogo ideale dove installare i due KP di tipo Aggregator, in quanto A3 ha una funzione centrale e quindi sarà

sempre attivo, mentre invece gli altri dispositivi possono essere spenti e accesi in qualunque momento.

Purtroppo questa macchina, il server dell'Arces, non è accessibile agli studenti, quindi per gli esperimenti i due aggregator verranno eseguiti sul dispositivo fisso A2.

Sugli altri dispositivi saranno in esecuzione due versioni dello stesso KP_QRCodeAdapter, sul PC con Windows 7 sarà sviluppato in C#, mentre sullo smartphone con Android in Java.

Il KP per il riconoscimento facciale sarà sviluppato solo in C# e quindi sarà disponibile solo per PC.

La Figura 4.4 riassume la situazione.

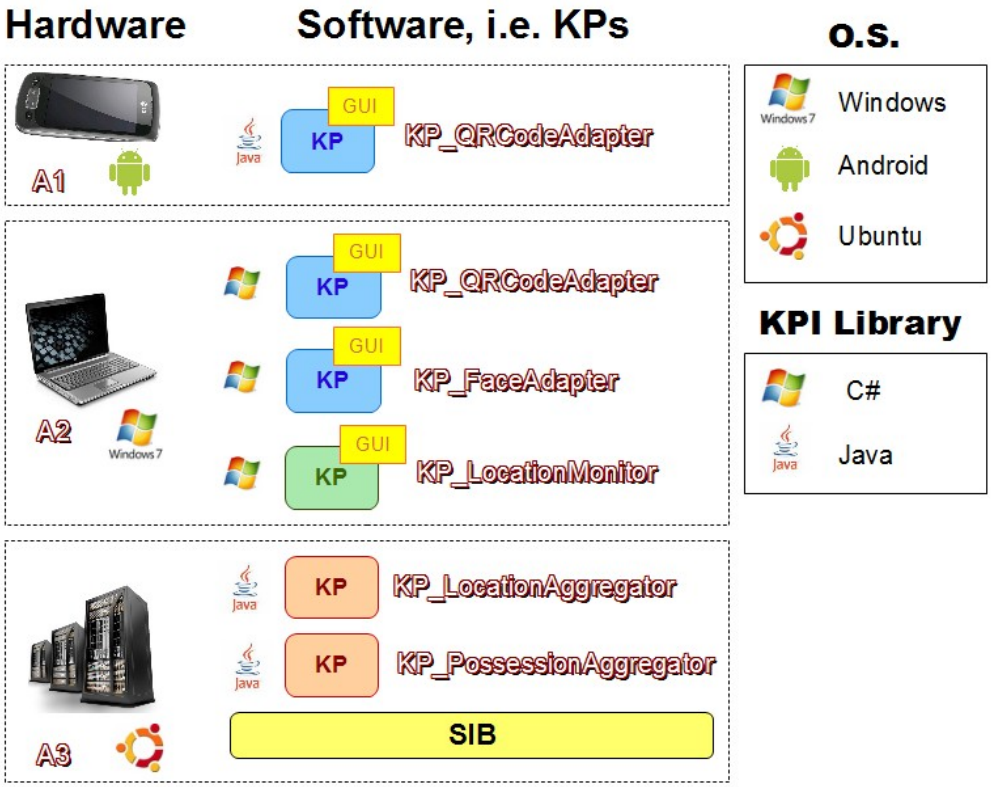


Figura 4.4: Infrastruttura Software-Hardware

L'ontologia e la tassonomia usate nell'ambito dell'applicazione sono simili a quelle già descritte nel capitolo 1.

Le entità presenti nell'ambiente intelligente sono i Device, gli User, gli Object e gli Space.

I Device possono generare dati dei sensori che vengono rappresentati attraverso la proprietà “HasSensorData” che ha come oggetto un'istanza della classe SensorData, a sua volta il dato ha la proprietà “HasMeasurand” che indica il tipo di misura eseguita, nel nostro caso o “Face” o “QR-CODE” e la proprietà “HasValue” che indica il valore effettivo misurato, ovvero il testo contenuto nel QR-CODE o l'Uri dell'immagine più somigliante ottenuta dal riconoscimento facciale. Inoltre per inserire l'informazione su quando è stato fatto il rilevamento il dato ha anche la proprietà “HasTimestamp” con oggetto un literal che esprime data e ora del rilevamento secondo un formato prestabilito.

Ogni entità invece può avere la proprietà “HasIdentificationData” che ha come oggetto un'istanza della classe “IdentificationData”. Similmente ai dati dei sensori quest'istanza avrà la proprietà “HasIdentificationType” analoga ad “HasMeasurand” e il valore del dato identificativo.

Quando il KP_LocationAggregator ricava correttamente le informazioni sulla localizzazione inserisce all'oggetto identificato la proprietà “HasLocation” con oggetto lo spazio entro il quale si trova e per quanto riguarda l'istante in cui è avvenuta la localizzazione si utilizza la proprietà “HasLocationTimestamp”.

La Figura 4.5 è un esempio di utilizzo dell'ontologia.

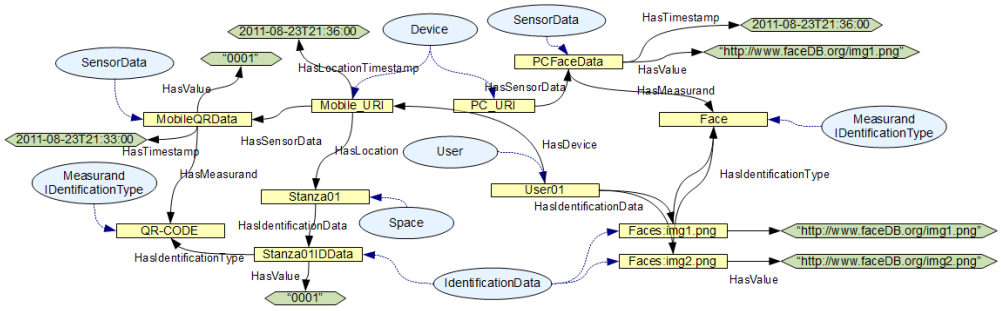


Figura 4.5: Ontologia

Continuando sulla linea guida descritta nel documento [34] è interessante definire meglio la porzione di ontologia di

competenza di ogni KP, in accordo alla filosofia di Smart-M3 di mantenere i KP semplici e separati tra loro.

4.2.3 KP_QRCodeAdapter

La versione Mobile di questo KP è sviluppata in Java in ambiente Android.

Le caratteristiche riassuntive sono descritte in Tabella 4.2, nella Figura 4.6 è rappresentata la parte di ontologia di competenza.

KP_QRCodeAdapter		
Executive summary		
F_ID	**Field Name**	**Value**
1_1	**Author**	Luca Feltrin
1_2	**Description**	QRCode Reader, produces sensor data
1_3	**Type**	Producer
Information Level and Semantic Interoperability		
2_1	**Ontology Awareness Required**	**Device** • HasFriendlyName • HasSensorData **Sensor Data** • HasValue • HasMeasurand • HasTimestamp
2_2	**Inserted Triples**	*<Device01 – HasFriendlyName - "Device01">* *<SensorData01 – HasMeasurand – QR-CODE>*
2_3	**Removed Triples**	-
2_4	**Updated Triples**	*<Device01 – HasSensorData – SensorData01>* *<SensorData01 – HasValue - "0001">* *<SensorData01 – HasTimestamp - "2010-09-15T12:14:00">*
2_5	**Queried Triples**	-
2_6	**Subscribed Triples**	-
System Level Portability		
3_1	**Program Language and KPI API**	C# * / Java **
3_2	**Runtime GUI**	yes
3_3	**SIB Version**	SIB-C Version 0.9.5 beta
*Note: *PC Version **Mobile Version*		

Tabella 4.2: Presentazione di KP_QRCodeAdapter

Da notare che in questa e nelle successive tabelle alcune triple evidenziate come "Updated Triples" vengono prima rimosse e poi reinserite piuttosto che semplicemente aggiornate. Questo perchè nel caso in cui un KP sia sottoscritto a quelle triple si vuole che la SIB notifichi in ogni caso

l'avvenuta update delle stesse, tuttavia usando la primitiva "update" la SIB notifica solo nel caso in cui cambi il contenuto della tripla e non quando rimane la stessa, comportamento indesiderato.

Le triple evidenziate come "Queried Triples", invece, sono quelle richieste da un KP per ottenere alcune informazioni al fine di eseguire il comportamento richiesto, in genere vengono richieste in fase di inizializzazione o durante alcune verifiche prima delle insert/update.

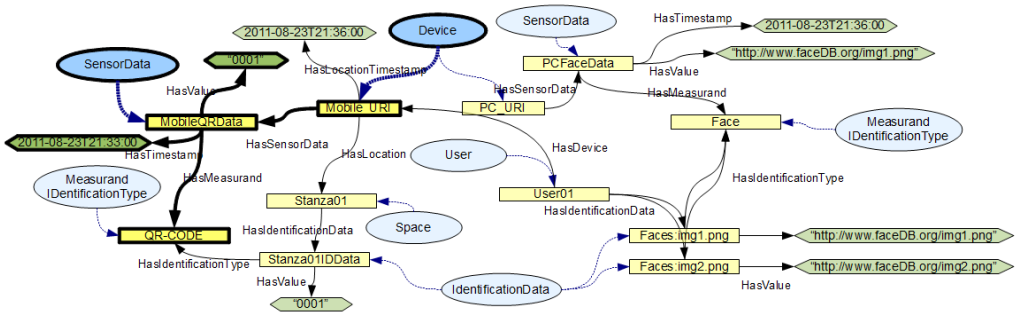


Figura 4.6: Parte del grafo di competenza di KP_QRCodeAdapter

Durante la fase di inizializzazione questo KP si occupa di creare le istanze del dispositivo sul quale è attivo e del SensorData corrispondente nel caso non esistano già.

Terminata questa fase ogni qual volta venga letto un nuovo codice viene modificato il valore del SensorData. Viene inoltre aggiornato anche il timestamp relativo a quel dato secondo un formato prestabilito.

Nell'interfaccia grafica è possibile inserire l'indirizzo IP della macchina sulla quale è in ascolto la SIB supponendo che di default la porta sia la 10010 e il nome dello Smart Space sia "X". Un pulsante permette di fare un breve test di connettività,

Figura 4.7: Interfaccia Grafica di KP_QRCodeAdapter versione Mobile

ovvero una semplice join che nel caso tutto funzioni bene restituirà true.

In una seconda casella di testo è possibile inserire in nome locale del dispositivo. Il KP concatena il namespace usato nell'ontologia, noto a priori, al nome locale e utilizza questo come Uri del dispositivo.

Premendo il pulsante “Start Scanning” l'applicazione lancia l'activity pre-installata sul dispositivo “Barcode Scanner” nel modo illustrato a pagina 29. La seconda activity esegue quindi la scansione del codice e ritorna alla prima il valore letto. Questo viene inserito nella SIB al posto del precedente valore.

La versione fissa è del tutto simile a quella mobile, a parte che la parte di riconoscimento del codice vera e propria è integrata all'interno del programma grazie alle librerie ZXing.

L'unica grossa differenza è l'integrazione con la libreria dotNetRDF: si è pensato di introdurre una nuova funzionalità, ovvero di codificare all'interno del QRCode un vero e proprio documento RDF/XML che contiene un piccolo grafo. Il KP prima di inserire il valore letto come SensorData, prova ad eseguire il parsing secondo la sintassi RDF/XML, se questa riesce allora inserisce tutte le triple contenute nel documento, qualsiasi esse siano, altrimenti prosegue come al solito.

Figura 4.8: Interfaccia Grafica di KP_QRCodeAdapter versione Fissa

Questo permette di realizzare nuovi meccanismi virtualmente senza limiti, per esempio supponendo che all'interno del sistema ci sia un KP che spedisce SMS e che entri in azione alla comparsa nella SIB di una certa tripla, questo sistema di invio può essere avviato grazie alla lettura di un QRCode che contenga un documento RDF con quella tripla.

In questo modo il KP_QRCodeAdapter diventa un producer general purpose che può essere usato per qualsiasi scopo.

Per creare un QRCode usando la sintassi RDF/XML è stata creata un'Utility, "RDF Triple to QRCode Converter" che permette di inserire una serie di triple e di convertirle in QRCode. L'utility ha bisogno dell'inserimento del namespace

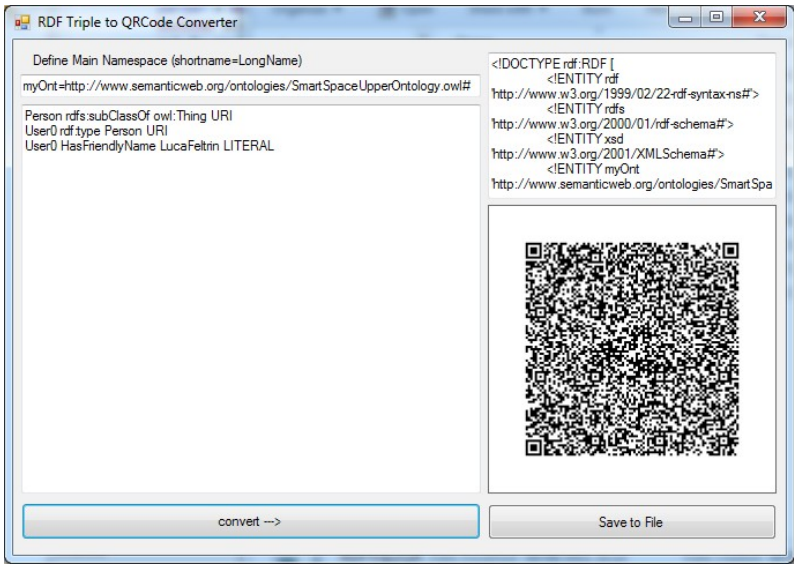


Figura 4.9: RDF Triple to QRCode Converter in funzione

da utilizzare e riconosce automaticamente i namespace noti come rdf, rdfs o owl.

Da notare che un documento RDF/XML che contiene anche solo poche triple è molto lungo, come si può vedere in Figura 4.9 il QRCode creato a partire da solo 3 triple è molto grande. Un codice del genere potrebbe non essere letto da un dispositivo dotato di webcam con bassa risoluzione. In sostanza è necessario considerare le capacità Hardware del dispositivo prima di usare codici così complessi.

4.2.4 KP_FaceAdapter

KP_FaceAdapter		
Executive summary		
F_ID	**Field Name**	**Value**
1_1	**Author**	Luca Feltrin
1_2	**Description**	Recognize User Faces and produces Sensor Data
1_3	**Type**	Producer
Information Level and Semantic Interoperability		
2_1	**Ontology Awareness Required**	**Device** • HasFriendlyName • HasSensorData **SensorData** • HasValue • HasMeasurand • HasTimestamp **User** • HasIdentificationData **IdentificationData** • HasValue • HasIdentificationType
2_2	**Inserted Triples**	*<Device01 – HasFriendlyName - "Device01">* *<SensorData01 – HasMeasurand – Face>*
2_3	**Removed Triples**	-
2_4	**Updated Triples**	*<Device01 – HasSensorData – SensorData01>* *<SensorData01 – HasValue -* *"http://imgServer/User01_1.png">* *<SensorData01 – HasTimestamp - "2010-09-15T12:14:00">*
2_5	**Queried Triples**	<User01 – HasIdentificationData – IdentificationData01> <IdentificationData01 – HasIdentificationType – Face> <IdentificationData01 – HasValue - "http://imgServer/User01_1.png">
2_6	**Subscribed Triples**	-
System Level Portability		
3_1	**Program Language and KPI API**	C#
3_2	**Runtime GUI**	yes
3_3	**SIB Version**	SIB-C Version 0.9.5 beta

Tabella 4.3: Presentazione di KP_FaceAdapter

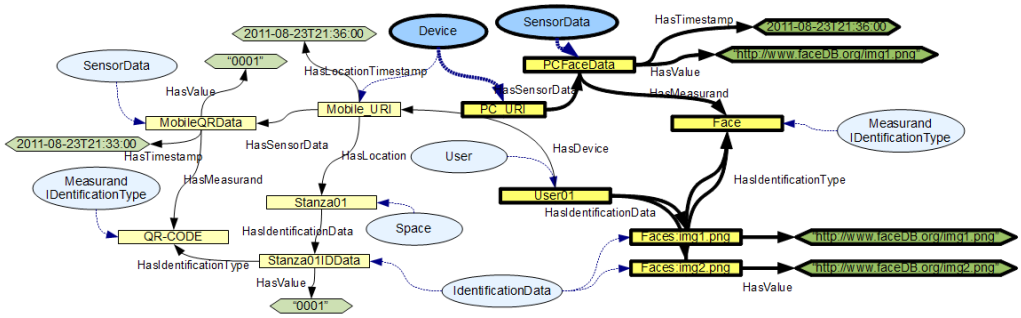


Figura 4.10: Parte del grafo di competenza di KP_FaceAdapter

Questo KP ha il compito di eseguire il riconoscimento facciale a partire dalle immagini della webcam, quindi di inserire il risultato dell'operazione all'interno della SIB analogamente al KP_QRCodeAdapter come dato di sensori.

Da notare che per mantenere omogeneità con il precedente KP, questo produrrà in uscita solo l'Uri dell'immagine più somigliante a quelle campione e non l'utente stesso, l'associazione tra immagine e utente, come anche tra QRCode e oggetto, sarà a carico degli aggregator.

KP_FaceAdapter è basato sull'algoritmo EigenFace, tuttavia qualora si volesse migliorare il sistema con altri algoritmi di riconoscimento facciale più sofisticati la natura modulare dell'applicazione permette una rapida sostituzione senza alterarne la struttura.

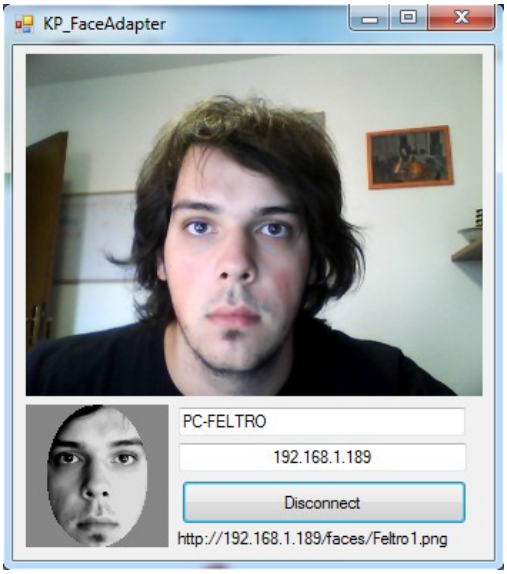


Figura 4.11: Interfaccia Grafica di KP_FaceAdapter

In fase di inizializzazione il KP esegue una Query per trovare tutti gli identificationData associati agli utenti di tipo faccia, ovvero recupera tutti gli Uri delle immagini campione degli utenti, quindi provvede a scaricarle e a pre-processarle, in caso il pre-processing restituisca correttamente un'immagine questa viene memorizzata all'interno dell'oggetto EigenObjectRecognizer.

Una volta creato l'oggetto il programma inizia ad acquisire i fotogrammi dalla webcam quindi prova ad eseguire il pre-processing, in caso di successo confronta l'immagine ottenuta con quelle campione e di nuovo in caso di successo inserisce nella SIB un SensorData contenente l'Uri dell'immagine più somigliante.

Per quanto riguarda gli aspetti più tecnici, il riconoscimento facciale e il pre-processing sono computazioni abbastanza pesanti che impiegano qualche istante per essere eseguiti. Se venissero eseguiti subito dopo l'acquisizione del fotogramma della webcam e subito prima della stampa a video dello stesso fotogramma l'utente vedrebbe l'immagine originale aggiornarsi a scatti, per questo motivo il riconoscimento viene eseguito da un Thread separato.

L'oggetto PictureBox presente nella GUI fa da buffer, il Thread principale vi inserisce il nuovo fotogramma mentre il Thread di riconoscimento quando è pronto lo legge ed esegue l'algoritmo.

Infine l'algoritmo di pre-processing, schematizzato in Figura 4.12, è stato sviluppato in modo empirico all'interno di una classe apposita che viene poi inclusa in tutti gli altri sorgenti.

Data l'immagine originale l'algoritmo cerca di trovare attraverso la tecnica delle Haar Features il volto e in seguito i due occhi.

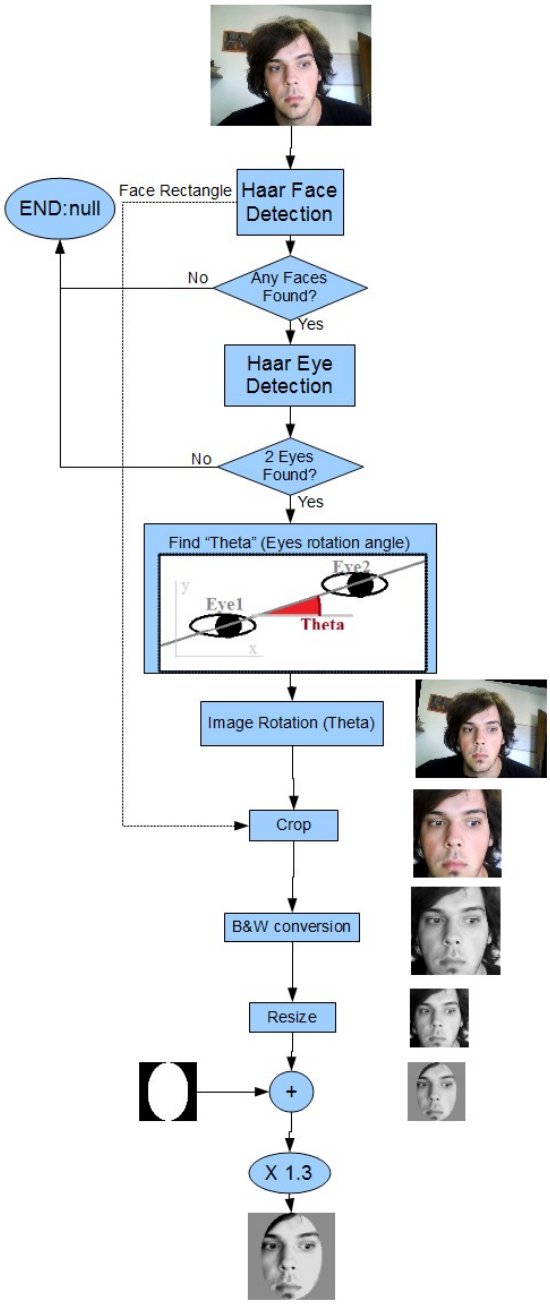


Figura 4.12: Algoritmo di Pre-Processing

In caso non vengano trovati l'algoritmo restituisce l'oggetto null. Il controllo sugli occhi ha il duplice scopo di ridurre la

probabilità di un falso riconoscimento del volto che capita spesso, e di rendere l'algoritmo complessivo meno dipendente dalla rotazione del volto. Infatti una volta che vengono trovate le coordinate degli occhi, viene calcolato l'angolo di rotazione del volto, quindi l'immagine viene ruotata dalla parte opposta mantenendo il volto perfettamente verticale.

In seguito l'immagine viene ritagliata e ridimensionata fino alla risoluzione standard 100x100, viene poi applicata una maschera ovale in modo da rendere l'algoritmo meno sensibile allo sfondo, e infine viene leggermente amplificata per esaltare le caratteristiche del volto.

4.2.5 KP_LocationAggregator

KP_LocationAggregator		
Executive summary		
F_ID	**Field Name**	**Value**
1_1	**Author**	Luca Feltrin
1_2	**Description**	Generates location data based on device's sensor data
1_3	**Type**	Aggregator
Information Level and Semantic Interoperability		
2_1	**Ontology Awareness Required**	**Device** • HasSensorData • HasLocation • HasLocationTimestamp • HasIdentificationData **SensorData** • HasValue • HasMeasurand • HasTimestamp **User/Object/Space** • HasIdentificationData • HasLocation • HasLocationTimestamp **IdentificationData** • HasValue • HasIdentificationType
2_2	**Inserted Triples**	-
2_3	**Removed Triples**	-
2_4	**Updated Triples**	<Thing01 – HasLocation – Space01> <Thing01 – HasLocationTimestamp - “2010-08-29T00:30”>
2_5	**Queried Triples**	<SensorData01 – HasMeasurand – QR-CODE|Face> <SensorData01 – HasValue - “sensor data value”> <IdentificationData01 – HasValue - “sensor Data value”> <IdentificationData01 – HasIdentificationType – QR-CODE|Face> <Thing01 – HasIdentificationData – IdentificationData01> <Thing01 – HasLocation – Space02> <Thing01 – HasLocationTimestamp - “2010-08-29T00:30”> <Device01 – HasLocation – Space02> <Device01 – HasLocationTimestamp - “2010-08-29T00:30”>
2_6	**Subscribed Triples**	<Device01 – HasSensorData – SensorData01>
System Level Portability		
3_1	**Program Language and KPI API**	Java
3_2	**Runtime GUI**	no
3_3	**SIB Version**	SIB-C Version 0.9.5 beta

Tabella 4.4: Presentazione di KP_LocationAggregator

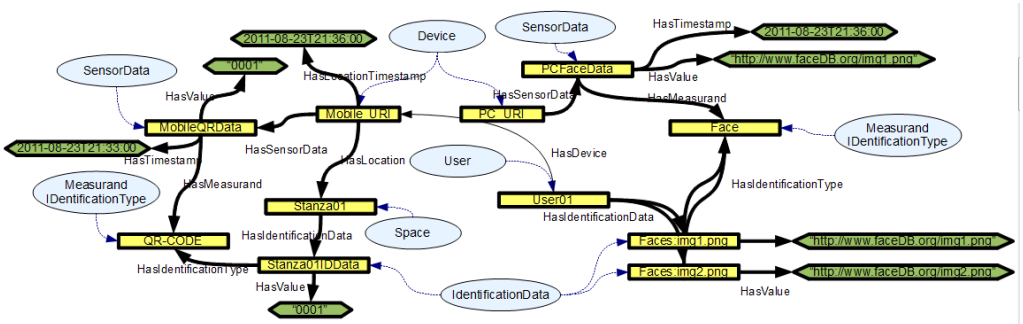


Figura 4.13: Parte del grafo di competenza di KP_LocationAggregator

Questo KP è il principale responsabile delle informazioni sulla locazione vera e propria.

È sottoscritto alle triple del tipo <Device01 – HasSensorData – SensorData01>, ovvero quelle prodotte dai KP che contengono i dati letti dai dispositivi.

Una volta ricevuta una nuova segnalazione dalla SIB, questo KP cerca tutte le entità che hanno un dato identificativo dello stesso tipo e dello stesso valore di quello letto dal dispositivo. Per ogni oggetto trovato si verifica se si tratta di uno spazio o di un altro oggetto.

Nel primo caso allora viene associata al dispositivo che ha compiuto l'identificazione la proprietà “HasLocation” con oggetto la stanza identificata.

In caso venga identificato un generico oggetto, che sia un dispositivo un utente o un oggetto inanimato, si possono verificare 4 situazioni diverse:

- Nessuno dei due, dispositivo e oggetto identificato, hanno dati di locazione, in questo caso nulla si può fare.
- Uno solo tra dispositivo e oggetto identificato possiede dati di locazione, in questo caso all'altro viene associata la locazione del primo.
- Entrambi hanno già dati di locazione, in questo caso il KP osserva il “LocationTimestamp” e fa in modo che all'entità con la locazione meno recente venga associata quella più recente.

4.2.6 KP_PossessionAggregator

KP_PossessionAggregator		
Executive summary		
F_ID	**Field Name**	**Value**
1_1	**Author**	Luca Feltrin
1_2	**Description**	Generates location data based on Possession informations
1_3	**Type**	Aggregator
Information Level and Semantic Interoperability		
2_1	**Ontology Awareness Required**	**Device/Object** • HasLocation • HasLocationTimestamp **User** • HasDevice • HasLocation • HasLocationTimestamp
2_2	**Inserted Triples**	-
2_3	**Removed Triples**	-
2_4	**Updated Triples**	<Device01|User01 – HasLocation – Space01> < Device01|User01 – HasLocationTimestamp - “2010-08-29T00:30”>
2_5	**Queried Triples**	<User01 – HasDevice – Device01>
2_6	**Subscribed Triples**	<User01|Device01 – HasLocation - Space01>
System Level Portability		
3_1	**Program Language and KPI API**	Java
3_2	**Runtime GUI**	no
3_3	**SIB Version**	SIB-C Version 0.9.5 beta

Tabella 4.5: Presentazione di KP_PossessionAggregator

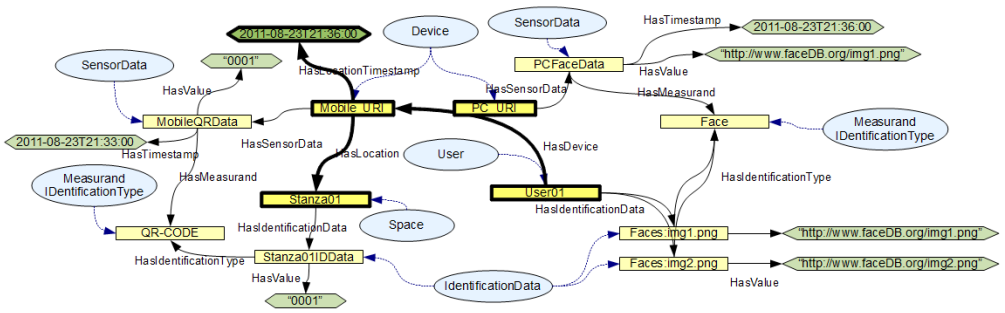


Figura 4.14: Parte del grafo di competenza di KP_PossessionAggregator

La maggior parte delle informazioni sulla localizzazione, e il riconoscimento stesso degli oggetti identificati, è realizzata dal precedente KP. Questo serve unicamente a ricavare qualche informazione in più basata sul fatto che un utente può possedere un dispositivo.

Il KP viene sottoscritto alle triple prodotte da KP_LocationAggregator, ovvero del tipo <any – HasLocation – any>.

L'oggetto di queste triple è ovviamente uno Space, il soggetto invece può essere qualsiasi cosa.

Nel caso in cui il soggetto sia un Utente, e nel caso questi abbia la proprietà HasDevice con oggetto un generico dispositivo, il KP localizzerà quest'ultimo nella stessa stanza.

Nel caso in cui il soggetto sia invece un dispositivo viene localizzato nella stanza l'utente che lo possiede.

L'effetto di questo aggregator è che un utente e un dispositivo legati insieme dalla proprietà HasDevice, viaggiano sempre insieme e si trovano sempre nello stesso luogo, e per cambiare location basta che venga identificato uno solo dei due.

4.2.7 KP_LocationMonitor

KP_LocationMonitor		
Executive summary		
F_ID	**Field Name**	**Value**
1_1	**Author**	Luca Feltrin
1_2	**Description**	Visualize all entities location
1_3	**Type**	Consumer
Information Level and Semantic Interoperability		
2_1	**Ontology Awareness Required**	**Device/Object/User** • HasLocation • HasLocationTimestamp • HasFriendlyName **Space** • HasFriendlyName
2_2	**Inserted Triples**	-
2_3	**Removed Triples**	-
2_4	**Updated Triples**	-
2_5	**Queried Triples**	<Thing01 – HasLocationTimestamp - “2011-09-15T12:41:00”> <Thing01 – HasFriendlyName - “Thing01”> <Space01 – HasFriendlyName - “Sala”>
2_6	**Subscribed Triples**	<Thing01 – HasLocation – Space01>
System Level Portability		
3_1	**Program Language and KPI API**	C#
3_2	**Runtime GUI**	yes
3_3	**SIB Version**	SIB-C Version 0.9.5 beta

Tabella 4.6: Presentazione di KP_LocationMonitor

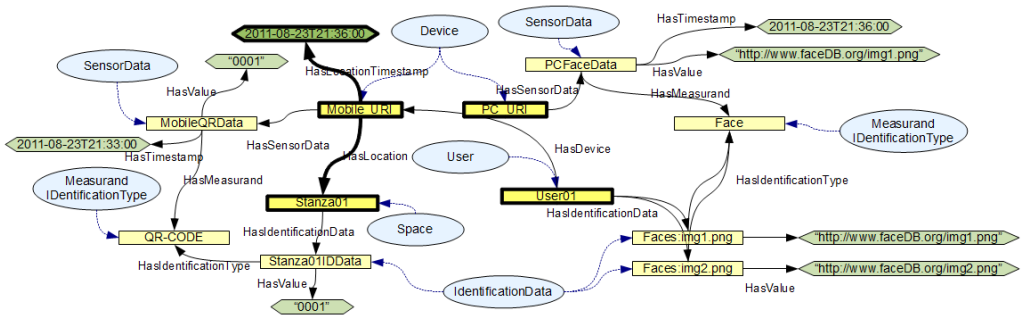


Figura 4.15: Parte del grafo di competenza di KP_LocationMonitor

Una volta che tutti gli altri KP hanno generato le informazioni sulla locazione degli oggetti all'interno dell'ambiente intelligente è necessario fornire agli utenti un'interfaccia grafica per poter visualizzare queste informazioni.

Lo scopo di KP_LocationMonitor è proprio questo, viene sottoscritto alle triple del tipo <any – HasLocation – any> e visualizza a schermo un elenco degli spazi trovati dentro la sib e per ognuno la lista degli oggetti contenuti all'interno con il timestamp relativo al loro ingresso nello spazio.

Nel caso in cui un'entità abbia la proprietà "HasFriendlyName" visualizzerà questa piuttosto che l'intero Uri.

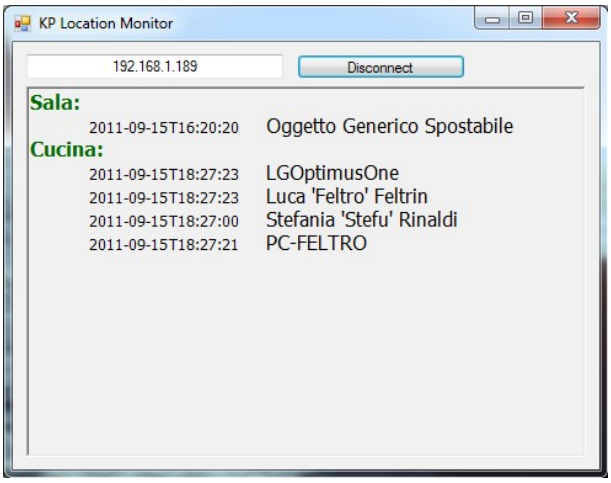


Figura 4.16: Interfaccia grafica di KP_LocationMonitor

Grazie a questo Software è stato realizzato il debug dell'intero sistema. In Figura 4.16 si può notare un'immagine dell'interfaccia grafica durante il funzionamento.

4.3 Collaudo del sistema

Per collaudare il sistema è stata elaborata una "sceneggiatura" ovvero una sequenza di azioni che simulano

una situazione tipo in vista anche della realizzazione di un video dimostrativo [36].

Dato che il PC è dotato solo di una webcam non è stato possibile testare contemporaneamente il KP_FaceAdapter e il KP_QRCodeAdapter versione fissa, quest'ultimo è stato testato a parte.

Due stanze sono state opportunamente taggate con i rispettivi QRCode come “Sala” e “Cucina”, il computer portatile (“PC-FELTRO”) sul quale sono in esecuzione gli Aggregator, KP_FaceAdapter e KP_LocationMonitor è stato posizionato in cucina e taggato, un oggetto generico (una caffettiera) è stata posizionata nella stessa stanza e taggata come “Oggetto Generico”.

Il primo utente, “Luca Feltrin” è quello dotato di Smartphone sul quale è in esecuzione KP_QRCodeAdapter, mentre invece “Stefania Rinaldi”, il secondo utente, ne è sprovvista.

1. Per prima cosa il primo utente entra in cucina ed esegue la lettura del codice identificativo della stanza utilizzando il suo Smartphone. KP_QRCodeAdapter riconosce il codice e produce nella sib il relativo SensorData, KP_LocationAggregator rileva questo dato, riconosce che il dispositivo ha identificato la stanza “Cucina” quindi asserisce che il dispositivo si trova in questa stanza. PossessionAggregator rileva quest'ultima asserzione e dato che lo Smartphone è in possesso di Luca Feltrin, localizza anch'egli in Cucina.
2. Luca Feltrin legge poi il codice a barre del Personal Computer. Anche questa volta lo Smartphone produce un nuovo SensorData che viene interpretato dal LocationAggregator come identificazione dell'oggetto “PC-FELTRO”. Dato che non è uno spazio e dato che lo Smartphone ha già una locazione, mentre il PC no, viene assegnato a quest'ultimo la locazione del primo, ovvero sempre la cucina.

3. Sempre nello stesso modo Luca Feltrin identifica anche la caffettiera che viene quindi a trovarsi nella stessa stanza, analogamente al punto 2.
4. A questo punto Luca Feltrin si sposta in sala leggendo, come richiesto dalle ipotesi, il QRCode della stanza. Si può notare che analogamente al punto 1 sia l'utente che lo Smartphone cambiano di locazione grazie al PossessionAggregator.
5. Infine per testare il FaceAdapter, StefaniaRinaldi entra in cucina e si fa riconoscere dal software. Una volta prodotto il rispettivo SensorData, LocationAggregator inserisce la locazione del secondo utente.

   Per il riconoscimento facciale è sono state inserite nell'ontologia, come IdentificationData di tipo Face, due immagini di identificazione degli gli utenti.

   Nel video si può notare come il software impieghi un po' di tempo ad eseguire il riconoscimento, inoltre in altri test è stato possibile verificare l'altra probabilità di errore dell'algoritmo. In futuro si potrebbe rivelare necessario studiare algoritmi più sofisticati di riconoscimento e creare quindi un nuovo KP più sofisticato. Si noti che, proprio a dimostrare questa tesi, questo upgrade è notevolmente semplice in quanto richiede la sostituzione di un solo componente, mentre gli altri KP possono continuare ad esistere, e a funzionare, in modo indipendente.

Per quanto riguarda invece la versione fissa del KP_QRCodeAdapter, durante il debug sono stati letti senza problemi più QRCode.

La funzionalità di inserimento di un documento RDF/XML codificato in un QRCode è stata testata utilizzando l'utilità descritta a pagina 58.

L'obiettivo è stato creare un QRCode che aggiungesse al sistema la nuova stanza “Studio”.

Il documento codificato conterrà le triple

```
<Studio – rdf:type – Space>
<Studio – HasFriendlyName – "Studio">
```

Evidentemente non è stato associato nessun dato identificativo alla stanza che quindi sarà inerte e non sarà possibile accedervi.

Il documento RDF/XML generato dall'utility è il seguente

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-
                                                ns#'>
  <!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
  <!ENTITY xsd 'http://www.w3.org/2001/XMLSchema#'>
  <!ENTITY LFOnt 'http://www.semanticweb.org/
        ontologies/SmartSpaceUpperOntology.owl#'>
]>
<rdf:RDF xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:LFOnt="http://www.semanticweb.org/
        ontologies/SmartSpaceUpperOntology.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <LFOnt:Space rdf:about="&LFOnt;Studio">
    <LFOnt:HasFriendlyName>Studio</LFOnt:HasFriendlyName>
  </LFOnt:Space>
</rdf:RDF>
```

Il Codice creato invece è mostrato in Figura 4.17.

Esso contiene solo due triple, tuttavia la lettura da parte del QRCodeAdapter è stata abbastanza difficoltosa, per esempio è stato necessario assicurarsi che il foglio fosse perfettamente in piano. Probabilmente QRCode con più di 4 o 5 triple sono impossibili da leggere con l'hardware a disposizione a causa delle dimensioni del documento XML.

Alla fine nel sistema comparirà una nuova stanza Studio come mostrato dalla Figura 4.18 .

Così come è stata inserita una nuova stanza sarà possibile inserire nella SIB qualsiasi altro tipo di informazione come anche nuovi utenti, oggetti o triple che fanno scattare eventi di qualche tipo, come l'invio di SMS in un KP sviluppato dal gruppo Arces.

Figura 4.17: Esempio di Documento RDF/XML codificato in un QRCode

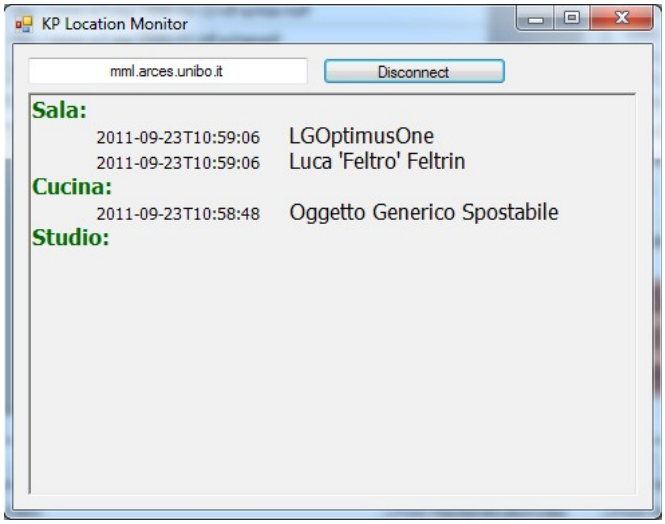


Figura 4.18: Interfaccia grafica di KP_LocationMonitor dopo l'inserimento della nuova stanza

5 Conclusioni

Va ricordato che l'esperimento realizzato è solo una delle possibili applicazioni che utilizzano l'identificazione, vero cuore dell'elaborato.

Lo scopo finale è di mostrare le caratteristiche peculiari di Smart-M3, ovvero l'interoperabilità tra diverse tecnologie, i vantaggi legati alla modularità delle applicazioni e la potenza della rappresentazione semantica dei dati, tutto ciò in un contesto basato sull'Ubiquitous Computing.

Nella parte sperimentale è già stato mostrato come integrare la tecnologia di Computer Vision con sistemi mobili e fissi come gli smartphone e i PC.

Si può pensare, in futuro, di estendere l'applicazione inserendo nuovi KP producer che inseriscono dati di identificazione basati su diversi tipi di tecnologie, per esempio il gruppo Arces ha già sviluppato KP dedicati alla lettura di tag RFID, questi KP possono essere già inseriti nel sistema e installati su Smartphone, PC o anche dispositivi embedded sviluppati ad hoc.

Ancora il sistema si può estendere creando altri KP che si occupino della lettura di dati biomedicali come per esempio le impronte digitali, oppure creando un KP consumer che si interfacci con un impianto domotico e che faccia accendere una luce, per esempio, quando la luce stessa viene identificata da qualcuno. L'adattamento tra tutte queste tecnologie avviene a livello del KP stesso installato sul determinato dispositivo che avrà come soli requisiti una porta di rete il fatto di essere programmabile e un opportuno interfacciamento col proprio hardware, ma Smart-M3 opera ad un livello di astrazione diverso da queste problematiche e quindi può interfacciare senza problemi qualsiasi hardware che rispetti i requisiti garantendo un alto grado di interoperabilità.

I vantaggi legati alla modularità delle applicazioni Smart-M3, e più in generale all'architettura a Spazio di dati condiviso, si vedono più che altro quando si ha la necessità di estendere il sistema o quando il sistema stesso cambia topologia in modo dinamico.

Infatti ogni KP, unità minima computazionale, ha una granularità abbastanza fine da permettere l'aggiunta di feature senza necessità di modifiche sostanziali a tutti i KP.

Nell'esempio descritto poco fa in cui si estende il sistema per identificare gli oggetti con altri tipi di tecnologie si fa notare che i KP Aggregator e il monitor rimarrebbero immutati in quanto il loro funzionamento non dipende dall'IdentificationType usato. Senza un'architettura di questo tipo modifiche del sistema richiederebbero un sostanziale cambiamento di ogni sua parte computazionale.

Probabilmente la vera potenza di Smart-M3, che rende questa tecnologia qualcosa di più rispetto a un semplice mezzo di coordinazione tra componenti, è il fatto che i dati sono rappresentati in modo semantico grazie a RDF e alle ontologie.

È già stato mostrato come sia possibile sfruttare la semantica per far fare ai KP semplici ragionamenti, che presi da soli sono piuttosto semplici, ma conferiscono al sistema nel complesso un comportamento intelligente.

Un esempio è nel KP_PossessionAggregator che esegue un ragionamento basato sul fatto che se A possiede B e A è in un posto allora lo è anche B.

Apparentemente il ragionamento è banale, ma pensando a sistemi e ontologie molto più complesse la capacità di ragionare anche in modo più sottile è molto importante.

Infine ripensando alla filosofia dell'Ubiquitous Computing, dove si hanno ambienti dotati di capacità computazionale invisibile agli occhi dell'utente, probabilmente non si può dire di essere pienamente soddisfatti in quanto il fatto di dover leggere un codice a barre ogni volta che si entra in una stanza o che si vuole compiere un'azione è tutt'altro che invisibile.

Tuttavia questo piccolo difetto è legato principalmente alla natura stessa della tecnica di identificazione usata che prevede un'esplicita azione da parte dell'utente che usando il proprio Smartphone deve leggere un codice a barre.

Il discorso cambia per il riconoscimento facciale, supponendo di migliorare l'algoritmo e migliorare l'hardware usato per avere l'input visivo, si può pensare a un ambiente in cui le telecamere sono nascoste e che quindi eseguono continuamente riconoscimenti senza che l'utente se ne renda conto. In questo caso l'informazione sulla localizzazione dell'utente sarebbe sempre presente senza che egli abbia compiuto una particolare azione.

Lo stesso risultato si può ottenere usando la tecnologia RFID a distanza piuttosto che i QRCode. Supponendo che l'utente possieda uno Smartphone dotato di lettore RFID a distanza si potrebbero posizionare i tag nei punti di accesso di ogni stanza, in questo modo la lettura avverrebbe al momento dell'accesso in modo totalmente invisibile all'utente.

In sostanza Smart-M3 è perfettamente in grado di funzionare secondo la filosofia degli ambienti intelligenti e di realizzare in futuro l'utopia di Mark Weiser dell'Ubiquitous Computing [12], in cui l'identificazione sarà sicuramente una funzione di primaria importanza.

6 Bibliografia

1 Haar-like features, *http://en.wikipedia.org/wiki/Haar-like_features*
2 QRCode, *http://en.wikipedia.org/wiki/QR_code*
3 Codici A Barre, *http://en.wikipedia.org/wiki/Barcode*
4 *F.Feltrinelli*, File2QR, an Android Application to Encode Files in QR Codes http://home.dei.polimi.it/bellasi/lib/exe/fetch.php?media=students:feltrinelli_finalreport.pdf
5 Denso-Wave: QRCodes, *http://www.denso-wave.com/qrcode/index-e.html*
6 Generatore di QRCode, *http://zxing.appspot.com/generator/*
7 Inpainting, *http://en.wikipedia.org/wiki/Inpainting*
8 Computer Vision, *http://en.wikipedia.org/wiki/Computer_vision*
9 *W. Zhao, R. Chellappa, A. Rosenfeld, P.J. Phillips*, Face Recognition: A Literature Survey 2003 http://www.face-rec.org/interesting-papers/General/zhao00face.pdf
10 Face Recognition Homepage, *http://www.face-rec.org/*
11 *M.Turk A.Pentland*, Eigenfaces for Recognition 1991 http://www.face-rec.org/algorithms/PCA/jcn.pdf
12 *Mark Weiser*, The Computer for the 21st Century
13 Namespaces in XML, *http://www.w3.org/TR/xml-names/*
14 RFC3986: URI Generic Syntax, *http://labs.apache.org/webarch/uri/rfc/rfc3986.html*
15 *T. Berners-Lee*, The Semantic Web
16 RDF, *http://it.wikipedia.org/wiki/Resource_Description_Framework*
17 OWL, *http://it.wikipedia.org/wiki/Web_Ontology_Language*
18 *F. Vergari, T. Salmon Cinotti, A.D'Elia, L.Roffia, G.Zamagni, C.Lamberti*, An Integrated Framework to Achieve Interoperability in Person-Centric Health Management
19 OpenCV, *http://en.wikipedia.org/wiki/OpenCV*
20 OpenCV project on SourceForge, *http://sourceforge.net/projects/opencvlibrary/*
21 Pagina ufficiale di Emgu CV, *http://www.emgu.com/wiki/index.php/Main_Page*
22 ZXing Pagina Ufficiale, *http://code.google.com/p/zxing/*
23 Android Developers Pagina Ufficiale, *http://developer.android.com/index.html*
24 dotNetRDF Pagina Principale, *http://www.dotnetrdf.org/*
25 RDF/XML Syntax Specification (Revised), *http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/*
26 *J. Soininen, P. Liuha, A. Lappetelainen, J. Honkola, K. Framling, R. Raisamo*, Device Interoperability: Emergence of the smart environment ecosystems 2010
27 Smart-M3, *http://en.wikipedia.org/wiki/Smart-M3*
28 *L.Roffia A.D'Elia F.Vergari D.Manaroli S.Bartolini G.Zamagni T.Salmon Cinotti J.Honkola*, A Smart-M3 lab course: approach and design style to

support student projects
29 Smart-M3 Pagina Ufficiale, *http://sourceforge.net/projects/smart-m3/*
30 API per Smart-M3 in C#, *http://sourceforge.net/projects/m3-csharp-kpi/*
31 API per Smart-M3 in Java, *http://sourceforge.net/projects/smartm3-javakpi/*
32 API per Smart-M3 in PHP, *http://sourceforge.net/projects/sm3-php-kpi-lib/*
33 *O.Lassila*, Semantic Web Programming using Piglet 2008
34 *L.Roffia, A.D'Elia, G.Vergari, D.Manzaroli, S.Bartolini, G.Zamagni, T.Salmon Cinotti, J.Honkola*, On the Design of Smart Space Applications
35 Imageshack, servizio di image hosting, *http://imageshack.us/*
36 Video dimostrativo su Youtube, *http://www.youtube.com/watch?v=JK0zX3I6YxI*