

Indice

1. Introduzione.....	7
2. Stato dell'Arte.....	11
2.1 <i>Ubiquitous Computing</i>	11
2.1.1 Esempi di Ubiquitous Computing	14
2.2 Il modello <i>Context-Aware</i>	16
2.2.1 Sistemi <i>Context-Aware</i>	18
2.2.2 Architettura.....	20
2.2.3 Il Context Model	24
2.2.4 Considerazioni sul modello <i>Ontology-Based</i>	26
2.3 Controllo dell'Accesso in ambienti Mobili e Pervasivi	28
2.3.1 Role-Based Access Control Model	30
2.3.2 Context-Aware Access Control Model	33
2.4 Applicazioni Mobili: piattaforme disponibili.....	35
2.4.1 Apple iOS	36
2.4.2 Symbian.....	37
2.4.3 BlackBerry.....	39
2.4.4 Windows Mobile	41
2.4.5 Google Android.....	42
2.4.6 La Piattaforma scelta: Android.....	44
2.5 Android ed il <i>Context-Aware</i>	45
3. Android.....	47

3.1 Introduzione ad Android	47
3.1.1 Architettura.....	50
3.1.2 Componenti Fondamentali	53
3.1.3 Interazione tra Componenti	55
3.1.4 Processi e <i>Thread</i>	57
3.2 Android vs iPhone	59
3.3 Android ed il <i>Semantic Web</i>	63
3.3.1 Il Semantic Web	64
3.3.2 Semantic Reasoner	64
3.3.3 Androjena	65
4. Progettazione	67
4.1 Requisiti Progettuali	67
4.1.1 Socially Aware Policy Model.....	71
4.1.2 Model Usability	75
4.2 Analisi del Sistema	76
4.2.1 Casi d'Uso	76
4.2.2 Scenari Operativi.....	78
4.2.3 Struttura Logica	78
4.3 Implementazione del Sistema.....	81
4.3.1 Graphic User Interface	83
4.3.2 Il <i>Semantic Reasoner</i> AndroJena	84
4.3.3 Policy Provider	84
4.3.4 Context Handler	87
4.3.5 Policy Enforcer Point	90
4.3.6 Policy Decision Point	91
4.3.7 Comportamento	92

5. Conclusioni e Sviluppi Futuri.....	95
Bibliografia.....	99
Ringraziamenti	101

1. Introduzione

La rapida diffusione di dispositivi mobili ha definito nuovi scenari operativi e modificato radicalmente le sfide che i progettisti devono affrontare. La creazione di un ambiente pervasivo ha di fatto posto al secondo posto problematiche prestazionali ponendo maggiore risalto a problematiche come eterogeneità, mobilità e dinamicità cui i dispositivi mobili devono necessariamente relazionarsi. L'ambiente di riferimento non è più un elemento statico o raramente modificabile, ma si tratta di un fattore estremamente variabile al quale il componente mobile deve sapersi adattare.

Il modello progettuale che meglio affronta questa sfida è quello *ContextAware*. Si tratta di sistemi che, potendo percepire l'evoluzione dell'ambiente circostante, tramite appositi sensori, sono in grado di reagire modificando il proprio comportamento di conseguenza. Analizzando la situazione di mercato è risultato evidente come gli smartphone siano i veri autori di questo cambiamento. Si tratta di apparecchi dotati di Sistema Operativo e una discreta capacità computazionale (che con il tempo si avvicina a quella di un computer) che però presentano vincoli strutturali tipici dei cellulari: batteria, memoria e costi di trasmissione dati sono i colli di bottiglia contro cui lottano i prodotti hardware e software di questo settore.

Vista la loro innata mobilità risultano essere ottimi componenti su cui testare modelli e teorie per risolvere le problematiche tipiche del pervasive computing. Tra quelle di maggior rilievo vi è la gestione del controllo dell'accesso. In un ambiente estremamente versatile le risorse disponibili cambiano rapidamente ed è necessario proteggerle da accessi non desiderati. Col termine risorsa possiamo indicare diversi elementi: file salvati sulla scheda di memoria, servizi e perfino l'attenzione dell'utente.

Scopo della tesi è la creazione di un sistema per il controllo dell'accesso di tipo Context e Social Aware che permetta di salvaguardare le risorse da accessi non desiderati. In particolare, una volta raccolte le informazioni sul contesto e sul chiamante, il sistema dovrà decidere, sulla base di politiche definite precedentemente, se sia lecito concedere o meno l'elemento richiesto.

La proposta di tesi prevede la realizzazione di un framework per la piattaforma Android (uno dei sistemi operativi per smartphone che si sta facendo largo nel mercato molto rapidamente) che permetta la definizione delle politiche di controllo dell'accesso e effettui il controllo dell'accesso sulla base di quest'ultime.

E' riservato particolare interesse alla modalità di interazione con cui l'utente può definire le politiche: essa dovrà rispecchiare il più fedelmente possibile il processo mentale di un utente medio evitando metodologie e procedure tipiche dell'amministratore di sistema. L'utenza telefonica infatti è caratterizzata da un livello culturale/informatico sensibilmente inferiore rispetto a quella dei notebook. Pertanto si vuole che chiunque possa essere in grado di interagire con il framework limitando al minimo indispensabile i tempi di apprendimento.

La tesi risulta suddivisa in tre parti. Nella prima si presenta lo Stato dell'Arte dei sistemi di controllo con particolare attenzione all'ambito mobile. Ci si sofferma a lungo sul modello Context Aware in quanto direttamente collegato alla realizzazione del progetto. Successivamente si analizzano le piattaforme disponibili, presentandone le caratteristiche essenziali e le motivazioni che hanno portato alla scelta finale.

Nella seconda parte il lettore è introdotto al cospetto di Android: sono delineate alcune delle caratteristiche di base del sistema operativo necessarie alla comprensione della fase di progettazione vera e propria. Unitamente a queste descrizioni vi sono elencate i punti salienti necessari a confrontare la piattaforma di Google con quella di Apple, iPhone.

Nel corso della terza parte si affronta la progettazione vera e propria. Vengono mostrate in dettaglio le fasi realizzative e inoltre spiegate le problematiche affrontate.

Nella sezione conclusiva vi sono ulteriori considerazioni sulle scelte implementative e gli eventuali sviluppi futuri.

2. Stato dell'Arte

In questo capitolo sono fornite le nozioni di base della Computazione Pervasiva, facendo attenzione all'applicazione nell'ambito dei dispositivi mobili.

2.1 *Ubiquitous Computing*

Ubiquitous (o anche *Pervasive*) *Computing* rappresenta il concetto di come gli utenti potrebbero utilizzare i supporti con capacità computazionali in futuro: non più interagendo con un computer alla volta, ma piuttosto interagendo con un set dinamico di computer interconnessi, spesso “invisibili” e integrati negli oggetti di tutti i giorni. Questo concetto è anche conosciuto come *Ambient Intelligence*, o *Smart Spaces*.

La locuzione “Ubiquitous Computing” fu utilizzata per la prima volta in ambito informatico da Mark Weiser (Weiser, 1991) (Xerox Palo Alto Research Center) in un famoso articolo: “The Computer for the 21st Century”. Nonostante siano passati quasi vent'anni dall'uscita di quella pubblicazione (corrispondenti a un'era geologica in ambito informatico), i concetti espressi sono considerati, a livello teorico, attuali e alla base dello sviluppo software e hardware degli ultimi anni.

Erano ancora gli anni '80 quando allo Xerox Research Center si cominciò a parlare di ubiquitous computing. In quel periodo la diffusione dei personal computer aumentava a ritmi vertiginosi: le stime approssimative degli apparecchi venduti erano di circa cinquanta milioni e già si categorizzava come un “oggetto di uso quotidiano”. L'innovazione tecnologica e i ritrovati tecnici avevano permesso la realizzazione di dispositivi di dimensioni inferiori e dal costo ridotto rendendoli di fatto accessibili a un'ampia fetta di mercato. Ma lo sviluppo

industriale e la diffusione non avevano in alcun modo intaccato il rapporto con questa tecnologia: l'interazione utente-macchina era rimasta pressoché identica a quella degli anni '60. Era ancora l'utente che doveva interagire direttamente e, cosa importante, consapevolmente con il dispositivo per sfruttarne le potenzialità e servizi.

L'obiettivo che si posero Weiser e i suoi colleghi era quello di scardinare proprio questo paradigma: superare e rivoluzionare il modo di rapportarsi con la tecnologia. Il computer del ventunesimo secolo doveva essere in grado di adottare pattern operativi totalmente diversi. E per fare ciò era necessario pensare alla tecnologia in maniera differente. Il PC non doveva più essere un costoso accessorio da aggiungere agli elettrodomestici di casa, ma doveva a tal punto diventare parte integrante della quotidianità da essere in grado di scomparire alla vista umana e nascondersi all'interno della vita di tutti i giorni.

Moltissimi personaggi in diversi ambiti hanno provato a dare una definizione di questo atteggiamento mentale (scienziati, economisti, filosofi, ecc...). Riassumendo in maniera semplificata le loro opinioni possiamo dire che: le cose ci sembrano scomparire all'interno della realtà stessa nel momento in cui le utilizziamo senza rendercene conto. La "scomparsa" predetta da Weiser non è altro che l'utilizzo quotidiano, inconsapevole e profondamente istintivo della tecnologia che risulta essere parte integrante dell'esperienza sensoriale. Come ci insegna l'esperienza è nel momento in cui una persona impara a utilizzare sufficientemente bene qualcosa che smette di pensare ad essa: chi abitualmente scrive al computer non pensa alla posizione dei tasti da premere. Semplicemente le sue mani sembrano conoscere già questa informazione.

Onde evitare fraintendimenti, il termine Ubiquitous Computing non indica il desiderio di realizzare un prodotto in grado di funzionare in qualsiasi condizione o di produrre un dispositivo dotato di enorme potenza di calcolo in grado di accedere a tutte le informazioni presenti sulla rete. In entrambi i casi avremmo sì dei prodotti straordinari, ma l'obiettivo preposto non sarebbe raggiunto poiché il modello di interazione sostanzialmente non sarebbe cambiato.

Weiser non voleva realizzare un computer che fornisse l'accesso a un mondo virtuale, ma desiderava esattamente l'opposto: "Un mondo dentro il computer". E per ottenere ciò ipotizzò che le interazioni macchina-uomo venissero totalmente ribaltate. Le azioni della vita quotidiana dovevano diventare degli input per il calcolatore e dovevano dunque avere una semantica specifica: non doveva essere più necessario sedersi davanti ad un monitor e utilizzare mouse e tastiera per ottenere un servizio. Bensì era sufficiente, spostarsi da una camera all'altra, accendere la televisione, aprire il frigorifero, spegnere la sveglia. Ogni azione umana diviene un'interazione con il sistema esattamente come il click del mouse o la pressione di un tasto della tastiera.

Per realizzare un ambiente di questo tipo sono necessari dispositivi autonomi ma allo stesso tempo in grado di interagire tra loro. Termostati, sensori luminosi, sensori acustici, sensori di posizione, elettrodomestici, ciascuno svolgendo il proprio compito contribuisce alla creazione di un ambiente fortemente pervasivo.

Tra i primi prototipi realizzati presso lo Xerox Center, segnaliamo "Active Badge". Si tratta di una I.D. Card in grado di rilevare la posizione di un utente all'interno dell'edificio e permettere in tal modo alcune operazioni: localizzazione, redirect delle chiamate, redirect del monitor del computer, etc. In questo particolare esempio lo spostamento fisico da una stanza all'altra è il modo in cui l'utente interagisce con il calcolatore. E ciò avviene in maniera del tutto inconsapevole e trasparente. L'impiegato non deve fare altro che fruire dei servizi messi a sua disposizione dal sistema.

Ciò che ci preme rilevare è che questo modello di interazione non necessita e non include rivoluzioni a livello di intelligenza artificiale. La creazione di un environment talmente intelligente da prevedere in nostri bisogni sicuramente una sfida per il futuro ma al momento è irrealizzabile.

Per ora ci si accontenta di poter disporre di un ambiente totalmente accessibile in grado di mettere in relazione i diversi ambiti della vita sociale. Un semplice esempio su tutti: si potrebbe avere la necessità di controllare il contenuto

del frigorifero. Se si avesse a disposizione un elettrodomestico in grado di fornirci questa informazione, ci si potrebbe collegare tramite *smartphone*, consultare la lista dei prodotti contenuti ed evitare di comprarne troppi o rimanerne senza. Ovviamente sono necessarie infrastrutture e tecnologie ad-hoc per creare un'applicazione che offra queste funzionalità. La realizzazione di un ambiente pervasivo comporta l'utilizzo di tanti componenti che devono essere economici, a basso consumo energetico e in grado di fornire autonomamente interazione utente-dispositivo e dispositivo-dispositivo.

Gli environment di questo tipo presentano pro e contro: se da un lato ci forniscono dinamicità, portabilità e semplificazione nell'esecuzione delle operazioni quotidiane, dall'altro si devono tenere conto e risolvere problematiche di sicurezza e privacy. I dati necessari al funzionamento di un ambiente di tipo pervasivo sono spesso informazioni sensibili (strettamente personali) e come tali devono essere salvaguardate.

2.1.1 Esempi di Ubiquitous Computing

Si consideri come esempio l'azione dello scattare di una fotografia. Fino agli anni '80, prima dell'avvento delle macchine digitali, era necessario assegnare manualmente tutti i parametri della macchina (tempo di apertura, messa a fuoco, ecc). Il salvataggio dell'immagine avveniva su pellicola che doveva essere completamente impressa per poter essere sostituita e che poteva contenere solamente 12 o 36 istantanee. Inoltre il rullino doveva essere consegnato a un laboratorio fotografico che, nell'arco di qualche giorno, avrebbe provveduto allo sviluppo e alla consegna delle foto e dei negativi.

Con l'avvento degli *smartphone* (Poslad, 2009), con fotocamera digitale inclusa, sono cambiati parecchi aspetti: in primis la gestione dei parametri dello scatto diviene automatica, le foto sono salvate su memorie interne e possono essere cancellate, spostate, modificate, etc. Inoltre il dispositivo può sfruttare delle applicazioni interne per aggiungere funzionalità una volta del tutto impensabili:

- “Taggare” una foto inserendo l’ora e la posizione ricavate tramite il servizio GPS.
- Utilizzare programmi di riconoscimento facciale e interfacciarsi con social network o database.
- Interfacciarsi direttamente a proiettori senza utilizzare cavi.
- Accedere a stampanti collegate alla rete senza passare da computer.
- Riprodurre musica e filmati.

L’utente può usufruire con un unico dispositivo di tanti servizi differenti che una volta erano forniti da apparecchi diversi. Ciò è possibile perché lo smartphone integra al suo interno le tecnologie necessarie all’interazione con il mondo esterno che, contestualmente, dovrà fornire il supporto ai servizi richiesti. Il tutto avviene senza che l’utente se ne renda conto. Nel momento in cui usiamo un singolo apparecchio in realtà accediamo a servizi diversi che sfruttiamo come se fossero parte integrante del medesimo dispositivo ma che in realtà è solo uno strumento per accedervi.

Un altro esempio rinomato è quello dell’*ubiquitous home environment*: consiste nella realizzazione di una casa intelligente con l’obiettivo di ridurre al minimo il consumo di risorse energetiche (acqua, luce, gas, ecc.). La gestione del riscaldamento è solitamente effettuata tramite termostato: la temperatura è impostata per fasce orarie o per tutta la giornata. Risulta scomodo modificare questi valori giornalmente in base agli impegni e ciò comporta che spesso l’impianto di riscaldamento sia attivo anche quando non siamo in casa.

Uno *smart environment* riesce, tramite appositi sensori di presenza e di temperatura, a gestire l’impianto di riscaldamento in maniera razionale evitando in tal modo sprechi energetici: quando stiamo dormendo (e quindi fermi in una sola stanza) abbassa le temperature delle altre stanze oppure se non siamo in casa si pone in modalità temperatura minima facendo in modo che non scenda sotto un certo valore. Per realizzare questo sistema sono necessari diversi dispositivi e sensori che dovranno interagire tra loro per decidere, in base alle preferenze

dell'inquilino e alle informazioni di contesto, come regolare i valori del funzionamento della caldaia o dei condizionatori.

2.2 Il modello *Context-Aware*

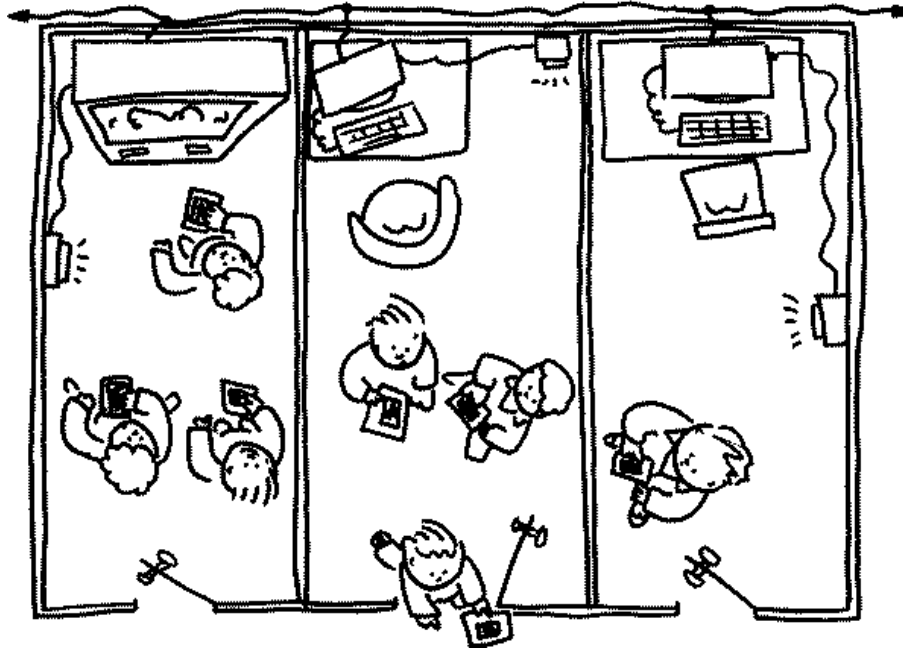
“Il contesto è qualsiasi informazione che può essere usata per caratterizzare lo stato di un'entità. Un'entità è una persona, un luogo, o un oggetto considerata rilevante per l'interazione tra un utente ed un'applicazione, incluso l'utente e le stesse applicazioni.” Dey and Abowd (Stojanovic, 2009) (Abowd, 2000)

La necessità di rispondere in maniera sempre più precisa alle richieste e ai bisogni degli utenti (soprattutto di quelli meno smaliziati) ha trainato vigorosamente lo sviluppo di sistemi intelligenti. La caratteristica principale alla base di questi ultimi è la capacità di adattarsi e reagire a seconda dell'evoluzione dell'ambiente circostante. Sono definiti Context-Aware in quanto effettuano le loro scelte di funzionamento non basandosi solamente su algoritmi o impostazioni interne come farebbe un normale programma, bensì facendo riferimento a un insieme di dispositivi/sensori che realizzano la loro parte “sensoriale”: è come se fossero in grado di avvertire l'evoluzione del mondo esterno.

Si può quindi comprendere come il modello Context Aware e quello Ubiquitous siano tra loro strettamente correlati: riprendendo l'esempio dell'ubiquitous home environment si nota che questo modello pervasivo prevede la presenza di un certo quantitativo di sensori i quali non sono altro che estensioni del sistema stesso: i dispositivi di localizzazione e di movimento sono gli occhi, i dispositivi per la rilevazione della temperatura, il tatto, etc. Per poter funzionare correttamente l'home environment necessita continuamente di dati che devono essere reperiti. Queste informazioni rappresentano il Contesto. Pertanto si tratta di un sistema di tipo ContextAware.

La locuzione “Context-Awareness in ubiquitous computing” venne introdotta per la prima volta da Schilit in un famoso articolo del 1994 intitolato “Context Aware Computing Application” (B Schilit, 1994).

Figura 1 Context-Aware Computer System



Egli proponeva un ambiente di lavoro dinamico in cui i dipendenti si potevano spostare da un ufficio all'altro con una sorta di palmare che, unito ad un sistema di rilevatori, permetteva di conoscere la posizione dell'utente all'interno dell'edificio. In tal modo era possibile visualizzare automaticamente sui monitor di un ufficio le informazioni e i programmi che un utente stava utilizzando in un'altra stanza. Oppure si potevano proiettare le informazioni di un certo progetto nel caso in cui un gruppo di progettisti si fosse riunito: il sistema avrebbe intuito che vi era un meeting in corso e dai profili degli utenti avrebbe compreso che probabilmente si stava discutendo di un progetto comune. Inoltre il sistema era in grado di localizzare la stampante più vicina per fare in modo che chi volesse stampare dei documenti non dovesse necessariamente ritornare al proprio ufficio o piano.

2.2.1 Sistemi *Context-Aware*

Ma cos'è che in definitiva differenzia un ambiente Context-Aware da uno non Context-Aware? Entrambi presentano un insieme di politiche o di direttive che regolano il comportamento del sistema a seconda delle richieste dell'utente, ma mentre per il secondo le politiche sono sempre valide, per il primo esse sono applicabili solo sotto certe condizioni.

Consideriamo questa politica: *Un utente può accedere ad una risorsa solamente se ci sono meno di 100 utenti attivi.*

Si tratta di una politica che viene utilizzata per garantire che il sistema non subisca arresti a causa di sovraccarichi: si definisce dunque un tetto massimo al numero di utenti contemporaneamente collegati. Questa politica verrà valutata ogni qual volta un utente richiede di sfruttare una risorsa. L'accesso in questo caso non dipende solamente dal ruolo o dai privilegi (elementi che caratterizzano staticamente un utente) ma anche da fattori esterni che l'amministratore ha deciso di valutare (numero totale utenti) e di cui il sistema dovrà tenere in considerazione.

Dunque un sistema context-aware permette di inserire dei livelli di dinamicità per quanto riguarda la gestione dell'accesso. Ovviamente ciò comporta dei costi non trascurabili. Infatti per poter attuare le regole è necessario utilizzare un quantitativo maggiore di risorse sia in termini di potenza di calcolo e sia in termini di complessità architetturale. Ciò è dovuto al fatto che è necessario reperire, volta per volta, le informazioni di contesto e utilizzarle nella fase di attuazione delle regole che risulta essere più complessa. Inoltre l'inserimento di livelli aggiutivi all'interno di un'architettura comporta inevitabilmente anche un aumento della probabilità di fallimenti e deadlock dei quali è necessario tenere presente e valutarne l'impatto sul funzionamento generale.

Un sistema Context-Aware ideale può essere rappresentato come un insieme di tre macro-funzionalità: *sensing*, *thinking*, *acting*. Ovviamente ciascun modello è differente dagli altri ma si può affermare con buona approssimazione che sono

queste le tre operazioni principali che deve svolgere. Esse possono essere definite come:

- *Sensing*: Un sensore, che sia di tipo biologico o elettro-meccanico dà la possibilità di recuperare un certo tipo di informazione. Ciascun sensore avrà una propria specifica area operativa. L'utilizzo di un numero elevato di sensori di tipologia diversa ci fornirà un quantitativo maggiore di informazioni su cui poter fare della thinking activity.
- *Thinking*: Il processo di valutazione delle informazioni provenienti dai vari sensori è una parte fondamentale del funzionamento del sistema. Questa parte si può considerare composta da due fasi logicamente distinte: assegnamento di un valore semantico alle informazioni raccolte e effettiva valutazione. La prima implica il fatto che il sistema sia in grado di dare un significato a ciò che riceve dal mondo esterno. E' la fase più complessa dell'intera applicazione in quanto il problema della semantica è qualcosa di ancora non standardizzato pertanto ancora non vi è una modalità univoca nell'approccio. Nel corso della seconda fase si effettua la vera e propria valutazione: una volta che sono a disposizione i dati forniti dall'utente e le informazioni del contesto, basandosi sull'insieme di politiche si ricava il comportamento.
- *Acting*: Una volta che le informazioni di contesto sono state recuperate o è stata riconosciuta la situazione in cui ci si trova può essere effettuato il vero e proprio enforcement: l'azione deve essere eseguita. Ovviamente come essa viene eseguita (prestazioni, interfacciamento con l'utente, etc...) è a discrezione del programmatore.

I prodotti software attualmente in commercio rispecchiano sostanzialmente questa struttura con le dovute approssimazioni: alcuni possono comprendere

sensori complessi e effettuare una fase di reasoning più rapida. Altri possono presentare una configurazione differente.

2.2.2 Architettura

Un sistema Context-Aware può essere implementato in molti modi diversi. L'approccio dipende principalmente dai requisiti, dai vincoli e dalle condizioni di funzionamento: localizzazione dei sensori (locali o remoti), numero dei possibili user (uno o più), risorse disponibili sul dispositivo (high-level PC o small mobile device) o necessità di dover espandere in seguito il sistema (M. Baldauf, 2004).

La metodologia di recupero delle informazioni di contesto risulta essere una delle variabili fondamentali e di importanza rilevante al momento della progettazione architetturale del sistema. In un articolo molto interessante "An Intelligent Broker Architecture for Pervasive Context-Aware System" (Chen, 2004), H. Chen ci presenta tre differenti filosofie progettuali per affrontare questo spinoso problema:

- *Direct sensor access*: Utilizzato quando i sensori sono locali al dispositivo. I drivers per potervi accedere sono all'interno dell'applicazione stessa ma ovviamente non è una metodologia progettuale applicabile a sistemi dinamici e di grosse dimensioni.
- *Middleware infrastructure*: è il modo più sensato e utilizzato per creare un sistema context aware. Obiettivo principale è separare la business logic dalla GUI. In pratica il livello applicativo non ha visione fisica dei sensori, ma utilizza semplicemente delle API messe a disposizione dal middleware.
- *Context server*: L'approccio logico direttamente successivo è quello di permettere l'accesso di più clients ai dati. In pratica si tratta di un'estensione dell'approccio a middleware introducendo un livello per la gestione dell'accesso remoto. Questa modalità è utile nel

momento in cui l'applicazione deve funzionare su dispositivi con poche risorse e che non possono fare affidamento su un'ampia potenza di calcolo.

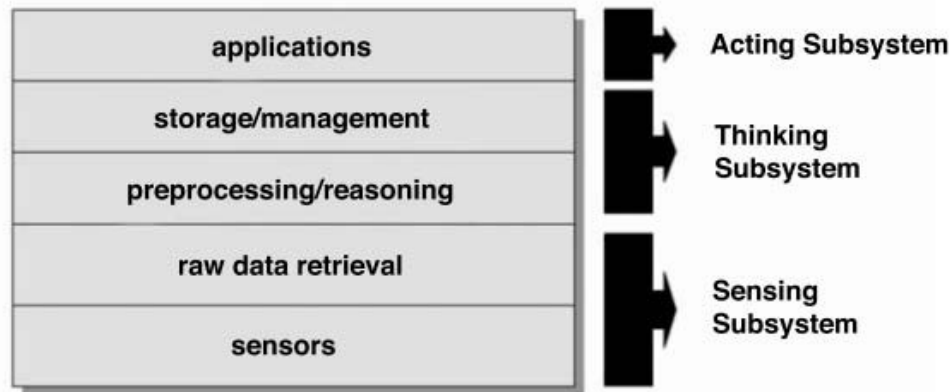
Per quanto riguarda il management del contesto per sistemi multiprocesso o multicomponente le scelte operative sono diverse. Winograd in "Architectures for context" del 2001 ce ne descrive tre possibili:

- *Widgets*: derivato dall'omonimo componente GUI è un componente software che fornisce un'interfaccia pubblica per un sensore hardware. Solitamente vi è un widget manager che gestisce tutti i widgets. Grazie all'incapsulamento e l'approccio "tightly coupled" questa modalità è efficiente ma non robusta per quanto riguarda la gestione dei fallimenti.
- *Networked services*: è il modello più flessibile. Invece di un global widget manager vengono utilizzate delle tecniche di discovery per trovare i servizi. In tal modo non è efficiente come potrebbe essere quello a widgets, ma risulta essere molto più robusto in caso di fallimenti.
- *Blackboard model*: contrariamente al process-centric model dei widget e il service-oriented model, il blackboard model rappresenta una visione data-centric. In questo approccio di tipo asimmetrico un client si registra affinché gli vengano notificate certe tipologie di eventi. Ciò fa sì che eventuali modifiche o aggiunte di eventi monitorati siano molto semplici da fare.

Ovviamente sono tante le proposte fatte in questi anni per realizzare strutture di questo tipo ma è ugualmente possibile definire un'architettura astratta che, con le dovute approssimazioni e semplificazioni, le rappresenta e riassume quasi tutte. Tratta da un articolo di Baldauf and Dustdar del 2004 questa

architettura astratta è composta da 5 layers con divisione in sottosistemi. Ciascun sottosistema si occupa di una delle tre macro-funzionalità (sensing, thinking, acting).

Figura 2 Architettura Astratta Context-Aware



Il primo layer (**Sensors**) consiste in una collezione di sensori. Da notare che il termine “sensore” non si riferisce esclusivamente a qualcosa di fisico, ma anche a tutte le sorgenti dato che possono essere utilizzate come contesto. In particolare i sensori possono essere catalogati in tre gruppi:

- *Physical sensors*: Sono quelli più utilizzati e al giorno d’oggi ne esistono tanti tipi in grado di raccogliere più o meno informazioni.
- *Virtual sensors*: Recuperano informazioni di contesto da software e da servizi. Ad esempio è possibile conoscere la posizione di una persona, oltre che tramite il GPS, anche controllando gli appuntamenti dell’agenda.
- *Logical sensors*: Combinano informazioni tratte da sensori fisici e virtuali per fornire dei dati più precisi ed accurati.

Il secondo layer (**Raw Data Retrivial**) si occupa del reperimento dei dati dai sensori tramite drivers per quelli fisici e tramite apposite API per quelli virtuali/logici. A questo livello dello stack il recupero delle infomazioni di

contesto avviene tramite l'invocazione di funzioni (Es. per ottenere la posizione geografica potrebbe esserci la funzione `getPosition()`)

Il livello di **Preprocessing** non è sempre implementato in tutti i sistemi context-aware ma è utile nel caso in cui i dati provenienti dai livelli inferiori siano eccessivamente specifici. Questo layer è responsabile per quanto riguarda il reasoning e l'interpretazione delle informazioni di contesto. In particolare il suo compito è di elevare il livello di astrazione dei dati provenienti dai livelli inferiori in modo da facilitare il compito a chi produce i livelli applicativi.

Le operazioni principali sono dunque quelle di aggregazione e composizione: solitamente il valore di un sensore non è così rilevante per un'applicazione che invece necessita di informazioni più elaborate. Queste due operazioni potrebbero essere svolte direttamente a livello applicativo, ma risulta essere più conveniente farle svolgere direttamente dal server: l'incapsulamento favorisce riusabilità e un più semplice sviluppo del livello applicativo.

E' possibile che si verifichi, quando si utilizzano molti sensori, il problema del Sensing Conflict: se si richiedono informazioni a diversi componenti è possibile che i dati che si ottengono siano tra loro contrastanti o addirittura diametralmente opposti. La soluzione ideale a questo problema deve essere realizzata a lato server in quanto solo lì si potrebbero avere informazioni aggiuntive (time-stamp, informazioni aggiuntive...) necessarie ad identificare quale delle due fonti sia la più affidabile.

Il quarto layer (**Storage and Management**) organizza i dati ottenuti e li offre tramite interfacce pubbliche ai client. Questi ultimi possono ottenere accesso in modalità sincrona o asincrona. In caso di accesso sincrono il cliente invoca dei metodi remoti e si sospende in attesa della risposta dal server. La modalità asincrona invece funziona tramite sottoscrizione: un client si registra per avere delle notifiche circa gli eventi riguardati alcuni valori del contesto di cui è interessato e nel momento in cui si verifica un dato evento gli viene notificato.

Il quinto ed ultimo layer (**Application**) è quello in cui viene realizzato il software client e l'enforcement vero e proprio. Si tratta del livello più alto che interagisce con l'utente o direttamente (ricevendo comandi) o indirettamente (mostrando informazioni tramite display).

2.2.3 Il Context Model

Il context model è uno strumento utilizzato per definire, salvare ed elaborare informazioni di contesto all'interno di una macchina. Nel corso degli anni sono state proposte tante modalità realizzative, ciascuna con vantaggi e svantaggi. Rifacendoci ad una pubblicazione di Stang e Linnhoff-Popien (Strang, 2004), presentiamo alcuni degli approcci più sfruttati per la modellazione del contesto: essi si basano su differenti strutture dati che vengono utilizzate per rappresentare e scambiare informazioni tra i vari attori del sistema.

- *Key-Value models*: modello più semplice per fare del context-modelling: una coppia key-value è utilizzata per descrivere le capacità di un servizio. L'operazione di discovery consiste in tal modo nell'utilizzo di un algoritmo che consulta queste coppie di valori.
- *Markup scheme models*: Si tratta di strutture gerarchiche composte da "markup tags with attributes and content". Esempi tipici sono: Composite Capabilities/Preference Profile (CC/PP) e User Agent Profile (UAProf) che sono codificati in RDF/S.
- *Graphical models*: L'Unified Modelling Language (UML) può essere utilizzato anche per fare del context-modelling. Ci sono diversi approcci di modellazione proprio in UML. (Vedi Sheng and Benatallah (2005)).
- *Object Oriented Model*: Utilizzare tecniche object-oriented per fare del context-modeling permette di sfruttare tutte le potenzialità del mondo OO (encapsulation, reusability, inheritance...). Approcci

esistenti utilizzano gli oggetti per rappresentare context type (come temperatura, posizione...) e incapsulano i dettagli del context-processing e della rappresentazione. L'accesso e le operazioni sul contesto sono disponibili tramite l'utilizzo di interfacce ben definite.

- *Logic based models*: Questa tipologia comporta un alto tasso di formalismo. Tipicamente “facts, expressions and rules” sono utilizzati per definire il modello del contesto. Nel mondo logic-based è possibile aggiungere nuovi fatti e fare del reasoning (derivare dei nuovi assunti) in base alle regole esistenti nel sistema.
- *Ontology based models*: Un'ontologia rappresenta una descrizione di concetti e di relazioni. Le ontologie sono ottimi strumenti per effettuare operazioni di context-modelling grazie alla loro espressività e alla possibilità di applicare tecniche di reasoning. Esistono attualmente diversi framework che usano delle ontologie per definire i loro context model.

Stang e Linnhoff-Popien, concludono affermando che le ontologie sono il modo migliore per fare del context-modelling in quanto risultano espressive e soddisfano la maggior parte dei requisiti richiesti. Alcuni dei requisiti e obiettivi alla base della rappresentazione di una context-ontology sono: semplicità, flessibilità ed estendibilità, genericità, espressività. Il problema è che definire ontologie in grado di definire contesti utilizzabili non è un compito facile. Per semplificare questo compito il World Wide Web Consortium ha proposto diversi standard per definire e condividere ontologie. Su tutti il Resource Description Framework (RDF, 2000) e il Web Ontology Language (OWL, 2004) (Manola, 2004) (AutoriVari, 2009).

Sono tante le discussioni in corso riguardanti problematiche di privacy e di sicurezza delle informazioni di contesto. Vi è grande interesse affinché queste informazioni rimangano riservate e non vengano utilizzate in maniera illecita. Vi

sono una serie di articoli e di proposte di ricerca che trattano di questo argomento: tra le possibili soluzioni si ipotizza l'introduzione di un toolkit. Tra i tanti proposti segnaliamo Confab che si prefigge lo scopo di aiutare l'utente nella gestione dei propri dati di contesto: a chi esporre queste informazioni, per quanto tempo, con che precisione. Si tratta quindi di un gestore in grado di definire delle politiche in modo da pubblicare solamente i dati che vogliamo esporre.

Inoltre le informazioni di contesto devono essere reperite rapidamente ed efficacemente altrimenti la scelta effettuata dal sistema potrebbe non essere più adeguata alla situazione attuale. Ad esempio si consideri un sistema wireless dipendente dalla posizione dell'utente. Se quest'ultimo si muovesse ad una velocità superiore alle capacità di recupero ed elaborazione, allora le decisioni prese del sistema non sarebbero coerenti con la situazione reale.

2.2.4 Considerazioni sul modello *Ontology-Based*

Nel corso degli ultimi anni sono stati realizzati una serie di framework che seguono sostanzialmente la struttura logica presentata precedentemente con l'obiettivo principale di separare chiaramente il livello applicativo dalle funzionalità Context-Aware. Tra i più famosi ricordiamo SOCAM (ServiceOriented Context-Aware Middleware), Gaia, CoBrA (Context Broker Architecture), the Context Toolkit, the Sentient Object Model, and Hydrogen.

Secondo il pensiero di Hong e Landay (2001) "un'infrastruttura è un insieme di tecnologie pubblicamente accessibili, affidabili, e pervasive che agiscono come base per altri sistemi" (J. I. Hong, 2001). L'idea è che tale tipo di supporto semplifichi significativamente lo sviluppo di applicazioni context-aware fornendo un set di API da utilizzare.

I vantaggi di questa modalità di progettazione in sintesi sono:

- Indipendenza dall'hardware, OS e linguaggi di programmazione: Formati di dato standard e protocolli possono essere utilizzati per accedere ai servizi offerti dall'infrastruttura da client eterogenei
- Miglioramento efficienza manutenzione e innovazione: I miglioramenti e bugfix per quanto riguarda le operazioni sul contesto sono realizzati a livello di infrastruttura e dunque indipendenti dall'applicazione. Per cui è necessario riferire agli user solamente le modifiche a livello di API.
- Condivisione di sensori, reasoning elaboration e servizi: Le risorse possono essere condivise da un numero elevato di processi facilitando in tal modo non solo la realizzazione di applicazioni ma anche la loro interoperabilità.

Ovvimente la parte di “sensing” è realizzata in maniera differente da framework a framework. Il fattore fondamentale è che sia incapsulata in modo tale da essere facilmente modificabile, espandibile e nello stesso tempo accessibile dall'esterno tramite API. Al momento non esiste un linguaggio di descrizione standard o un'ontologia per quanto riguarda le informazioni di contesto che possano essere riutilizzata su più middleware o framework.

A causa di questa mancata standardizzazione sono state presentate diverse soluzioni proprietarie. SOCAM, per esempio, utilizza l'approccio più sofisticato per la gestione delle informazioni di contesto: i sensori esterni sono acceduti tramite web services (usando SOAP, Simple Object Access Protocol), mentre all'interno gli eventi del contesto sono rappresentati in OWL con un'ontologia predefinita.

Context Model e Context Processing Logic sono tra i fattori di scelta più influenti nel momento in cui si elabora l'architettura del sistema. Un modello basato su ontologie e dotato di un motore in grado di effettuare del reasoning ha la capacità di derivare nuovi concetti per poter adattare il comportamento del sistema stesso. Un modello che non fa uso di ontologie, come ad esempio il “Context-Toolkit” che si basa su tuple, necessita di un pesante lavoro di

programmazione in quanto questi attributi non hanno di per sè una semantica in quanto facenti parte di alcun modello. E' il programmatore che tramite l'elaborazione assegna un significato a questi valori. Ciò comporta anche il fatto che l'assenza di semantica non permette a reasoner di ottenere nuove informazioni nel caso in cui vengano aggiunte o eliminate regole. Ovviamente gli aspetti di interesse in questi ambiti sono realmente tanti. Per il momento ci accontentiamo di sottolineare al lettore come una tipologia di approccio possa modificare radicalmente la realizzazione di un prodotto.

2.3 Controllo dell'Accesso in ambienti Mobili e Pervasivi

Il controllo dell'accesso è quel processo che concede o nega permessi ad un richiedente, sulla base di politiche, per accedere ad una determinata risorsa.

Lo sviluppo nelle applicazioni Mobili e Pervasive (Ubiquitous) stanno mettendo alla prova le classiche soluzioni proposte in quanto l'operare in un ambiente estremamente variabile ed eterogeneo, fa sì che il bisogno di proteggere le risorse da accessi indesiderati sia quanto meno indispensabile. I nuovi pattern operativi vertono sull'utilizzo di informazioni di contesto per la definizione di politiche e per la loro attuazione in quanto offrono maggiore dinamicità e flessibilità, caratteristiche vitali per questi ambienti. L'avere a disposizione dispositivi a basse risorse energetiche (low-power devices), comporta che la realizzazione e l'utilizzo di un sistema di controllo dell'accesso testato su computer risulti essere ancora troppo oneroso per apparecchi a risorse così limitate (L. Gomez, 2009).

Oltre alla necessità di utilizzare comunicazioni criptate a protezione dei dati in transito e di effettuare l'enforcement, tutti i meccanismi di controllo dell'accesso richiedono di autenticare l'utente. Tradizionalmente ciò viene fatto tramite una combinazione di fattori staticamente predefiniti all'interno delle

politiche di controllo dell'accesso del servizio di autorizzazione. Ma nell'ambito Mobile e Ubiquitous c'è il chiaro bisogno di una autenticazione utente flessibile basata non più su fattori statici, ma bensì su elementi disponibili in quel determinato momento, ed è questa la direzione verso cui la tecnologia si sta muovendo.

Si è discusso di Ubiquitous Computing definendolo come uno smart-environment in cui gli oggetti quotidiani diventano strumenti dalle capacità computazionali. Si è visto come l'utilizzo di informazioni di contesto rappresenti un significativo beneficio per applicazioni in ambiti estremamente dinamici. Il deployment di componenti che, nell'ambiente Ubiquitous, devono collaborare fa sì che la domanda di sicurezza aumenti quotidianamente. Inoltre è necessario proteggere le risorse condivise anche da accessi contemporanei di più applicazioni oltre che ovviamente da applicazioni esterne al sistema. Il controllo dell'accesso rappresenta una sfida complessa a causa soprattutto dell'elevata dinamicità del sistema in cui partner sconosciuti devono comunicare per scambiarsi informazioni.

Access Control è una tecnica di sicurezza standard per preservare e controllare le risorse del sistema. Essa consiste di una serie di meccanismi e processi che permettono la definizione di regole di accesso, più propriamente denominate "politiche", e di un sistema di enforcement per queste regole.

Access Control è il processo che garantisce un permesso in accordo a delle politiche. Una politica solitamente è del tipo "*Who can do what to what*". Dove il who è il soggetto, il primo what è l'action e l'altro what è la risorsa. In un sistema di tipo context-aware, il contesto è considerato come una condizione aggiuntiva. "*Who can do what to what under which circumstances*", le circostanze coincidono con il contesto dell'applicazione.

La disponibilità del contesto dell'applicazione permette un'impostazione ed una riconfigurazione del sistema e delle applicazioni di sicurezza dipendenti dal cambiamento del contesto.

Con *Context-Aware Security* si intende un enforcing dinamico delle politiche in base alle fluttuazioni dei valori di contesto.

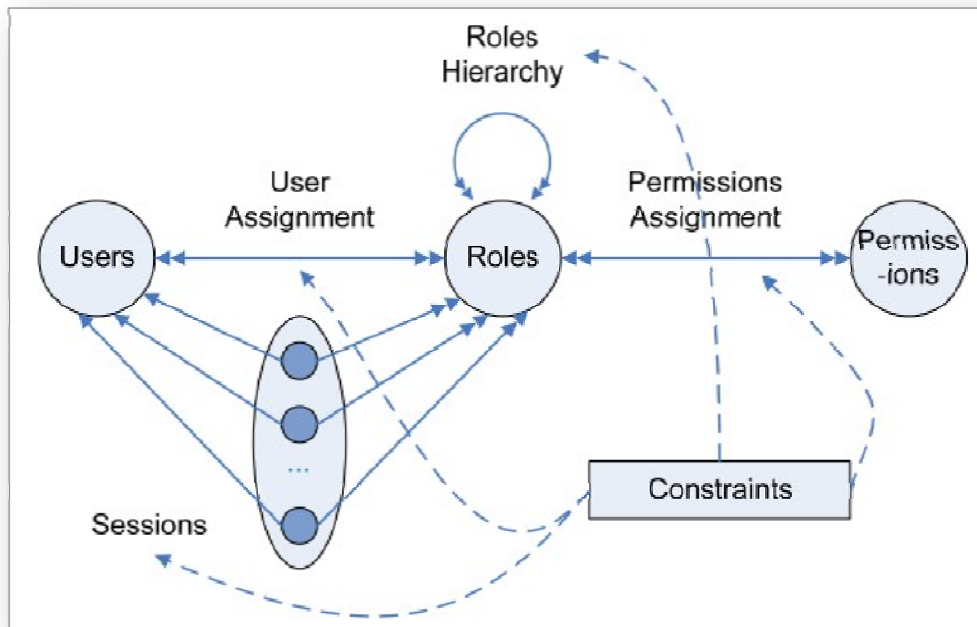
Ad esempio le informazioni di contesto possono essere utilizzate per riconfigurare i meccanismi di sicurezza per garantire maggiore livello di protezione e nel medesimo istante per ottimizzare il consumo di risorse del sistema. Ad esempio se dobbiamo mandare un'email tramite un hotspot pubblico il sistema automaticamente cripterà il messaggio, mentre se utilizziamo quello di casa o dell'ufficio, per risparmiare risorse il messaggio potrà essere inviato in chiaro.

Nei paragrafi successivi si presenterà il modello, per i sistemi di controllo, che meglio si adatta al mondo Mobile: il Context-Aware (E. Reshetova, 2008).

2.3.1 Role-Based Access Control Model

Apparso per la prima volta intorno agli anni '70, si basa sostanzialmente sul presupposto che ad ogni soggetto è assegnato un ruolo e ogni ruolo ha i propri permessi. I ruoli sono ovviamente creati in base alle esigenze del sistema. Dunque il modello si compone di tre entità: Users (U), Roles (R) e Permissions (P). Lo schema seguente sintetizza meglio di qualsiasi descrizione il funzionamento logico del sistema.

Figura 3 Modello Role-Based



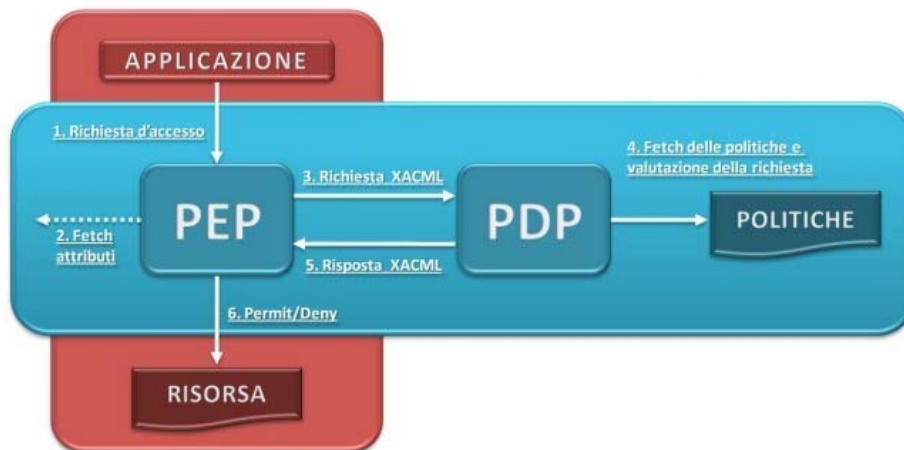
Il sistema viene realizzato sulla base di due diverse tipologie di assegnamenti: user assignment (UA) e permission assignment (PA) che sono entrambe relazioni multi-a-molti. Ad un user possono essere assegnati differenti ruoli e a ciascun ruolo possono essere assegnati molteplici user. Lo stesso identico ragionamento vale nel rapporto ruoli-permessi. La parte finale del modello considera il concetto di Sessione (S): uno user attiva un determinato ruolo durante una specifica sessione di lavoro. Di fatto uno user può attivare più di un ruolo simultaneamente in una sessione, ma una sessione può essere assegnata ad un singolo user.

Tra le implementazioni più recenti di RBAC troviamo XACML (Moses, 2005). Acronimo di eXtensible Access Control Markup Language, si tratta di un linguaggio dichiarativo per policy di controllo dell'accesso implementato in XML e di un processing model che descrive come interpretare le politiche.

Gli elementi che compongono il processing model sono:

- Policy Administration Point: Point che gestisce le policies.
- Policy Decision Point – Point che valuta e prende le decisioni circa il comportamento da applicare.
- Policy Enforcement Point - Point che intercetta la richiesta di accesso ad una risorsa da parte di un utente e mette in atto la decisione del PDP.
- Policy Information Point - Point che ha il compito di recuperare informazioni esterne necessarie all'enforcement.

Figura 4 Comportamento Logico XACML



Una Policy è strutturata come un'insieme di Rules che si compongono di Subjects, Resources, Actions. Il Subject è l'elemento che tenta l'accesso ad una risorsa e può avere uno o più attributi. La Resource è quel componente del sistema (dato, servizio, processo) che viene richiesto e può avere un singolo attributo. Una Action definisce il tipo di accesso richiesto per una risorsa e può avere diversi attributi. Vi è un elemento addizionale, Environment, che serve per fornire informazioni addizionali. Per il funzionamento nel dettaglio rimandiamo alle varie implementazioni disponibili in rete.

2.3.2 Context-Aware Access Control Model

Il modello Context-Aware può essere considerato come un'estensione del modello Role-Based Access Control con l'inserimento di informazioni aggiuntive. I sistemi RBAC infatti mostrano tutti i loro limiti nel momento in cui il controllo diviene eccessivamente complesso ed è necessario aggiungere informazioni addizionali all'enforcement process. Le soluzioni proposte per migliorare RBAC ovviamente sono svariate e offrono visuali e prospettive tra loro differenti.

Tra le idee più recenti abbiamo OrBAC (Organization-based access control) (Kalam, 2003). Come suggerisce il nome OrBAC fa uso di informazioni contestuali riguardanti struttura organizzative.

Altre possibili estensioni sono Proximity-based access control che definisce un set di regole basandosi sulla prossimità di gruppi o di entità (Gupta, 2006). Temporal-RBAC (Bertino, 2001) supporta un gestione tempodipendente delle politiche di accesso al sistema introducendo il fattore "tempo" come una variabile determinante nella gestione della sicurezza. L'Encounterbased access control (Thomas 2004) è utilizzato per definire policy in relazione a definite situazioni o eventi.

Il Generalized Role-Based Access Control (GRBAC) (Convington, 2002) propone un framework per fornire un controllo dell'accesso uniforme e poter definire environments roles: anche in questo caso si tratta di un'estensione del RBAC. In un certo dominio amministrativo, un ruolo può essere, per esempio, impiegato o manager. Un ruolo determina la posizione di un utente o il livello all'interno di un certo dominio amministrativo. Un environments roles è un ruolo che comprende anche le condizioni dell'environment. A differenza del RBAC, che è solo subject-oriented, il GRBAC permette la definizione di politiche basate su subject, object o environment.

Il Dynamic Role-Based Access Control (DRBAC) (Zhang 2004) estende il tradizionale RBAC all'utilizzo di informazioni di contesto dinamiche nel corso del processo di enforcing. Questo modello presenta due spunti chiave:

- I privilegi di accesso di un utente devono cambiare nel momento in cui le informazioni di contesto variano.
- Una risorsa deve modificare le proprie regole di accesso quando le informazioni di sistema variano.

Un sistema per il controllo dell'accesso di tipo context-aware può essere visto secondo due differenti prospettive:

- Adattamento delle politiche di sicurezza sulla base del contesto.
- Acquisizione sicura delle informazioni di contesto.

La prima si riferisce al fatto che un sistema è considerato Context Aware se utilizza le informazioni di contesto nel corso della fornitura del servizio. La seconda mette in risalto il fatto che questi dati devono essere prelevati in modo tale che si possa fare affidamento sulla loro veridicità.

MOSQUITO: si tratta di un framework progettato e implementato nel 2006 per mobile application basato su web services. Tutti i messaggi SOAP tra le applicazioni (web service client) e il web service stesso, devono attraversare degli intermediari, per far applicare la politica di sicurezza predefinita dal livello del messaggio. Gli intermediari sono una "pipeline of message filters" (proxy) che supportano WS-Security (OASIS, 2006). Gli intermediari clientside si occupano della cifratura, del controllo dell'integrità e delle credenziali. Lato server effettuano sostanzialmente le stesse operazioni: decifratura e controlli di integrità e di autenticità.

Il Security Ticket Service (STS) (OASIS, 2005) lato client genera informazioni di contesto firmate recuperate dal Context Information Provider (CIP). Lato server vengono prelevate queste informazioni del messaggio SOAP, viene verificata la firma e il PDP decide poi come agire in base alle politiche definite. Manderà ovviamente un messaggio di risposta firmato al client.

Le politiche sono salvate in XACML che è un sistema di tipo RBAC con estensione per la gestione delle informazioni di contesto.

Questo tipo di architettura, così fortemente basata su web services, è molto facilmente applicabile a sistemi di tipo mobile e ad ambienti pervasivi: basandosi sul Service Oriented Approach (SOA) il sistema risulta essere infatti sufficientemente flessibile ed efficiente. Inoltre questa tipologia di approccio dimostra come un sistema RBAC standard possa essere esteso per gestire informazioni di contesto.

2.4 Applicazioni Mobili: piattaforme disponibili

Obiettivo della tesi, come verrà analizzato nella parte relativa alla progettazione, riguarda il controllo dell'accesso su dispositivi Mobili.

Diventa quindi di fondamentale importanza scegliere accuratamente la piattaforma di sviluppo in grado di permettere la progettazione di un framework per il Controllo dell'Accesso *Context-Aware* caratterizzato dal modello di contesto *Ontology-Based*.

Di seguito elenchiamo le possibilità offerte dal mercato con una breve descrizione degli aspetti salienti.

2.4.1 Apple iOS

Uno dei dispositivi più diffusi del mercato, l'iPhone vanta nel suo "AppStore" circa 200.000 applicazioni.

Per svilupparne una occorre:

- Un qualsiasi computer Mac Intel-based
- L'ambiente di sviluppo gratuito XCode (presente direttamente nel DVD di OSX)
- L'iPhone / iPod Touch SDK, scaricabile gratuitamente
- Le chiavi per firmare digitalmente per proprie applicazioni: 99\$ l'anno

Conoscenze richieste: Objective-C & Cocoa Touch. Objective-C è molto differente dal "C" originale, rimane invariata solo la sintassi degli statement di base (if, for, while, switch ecc.). Definizione di interfacce, classi, UI ecc. risultano nuovi agli sviluppatori non abituati alla programmazione per Mac. Viceversa, chi già è in grado di programmare applicazioni desktop per OSX si sentirà piuttosto a suo agio: stesso strumento per creare le interfacce utente (Interface Builder), stesso linguaggio, e framework simile (Cocoa Touch, un sottoinsieme di Cocoa). Ovviamente per chi viene da altri mondi (C#/VB.NET su Windows, o magari Java) l'impresa non è certo impossibile, ma solo un po' più ardua rispetto ad altre piattaforme.

Pro:

- Presenza del DB engine Sqlite, e di una versione ridotta di OpenGL
- Uniformità dei dispositivi: si sviluppa una volta sola, sapendo che tutti i propri utenti avranno dei device praticamente identici (stessa risoluzione, stessa potenza, stessa risoluzione della fotocamera ecc.).

Contro:

- Gli strumenti di sviluppo sono gratis, ma bisogna per forza avere un Mac per sviluppare.
- Sono necessari 99\$ all'anno per avere la possibilità di pubblicare nello store di Apple
- E' necessario pagare la licenza anche per applicazioni personalizzate per uno specifico cliente. Anzi in quel caso bisogna spendere 299\$ per la licenza Enterprise.
- La documentazione e il supporto (sotto forma di forum, chat ecc.) sono ancora limitati rispetto ad altre piattaforme di sviluppo

Sebbene il sistema operativo di iPhone abbia molte potenzialità, di fatto poi le API pubbliche consentono di fare molto meno: risulta impossibile fare applicazioni che eseguono in background, applicazioni che intercettano eventi di sistema (chiamata / sms in arrivo), inviano SMS o mail, interagiscono con applicazioni di sistema quali il calendario, e che in generale accedono al device a basso livello. Le API per tutto ciò ovviamente esistono e Apple le usa per i propri applicativi, ma sono classificate come "private", non sono documentate.

2.4.2 Symbian

I device con Symbian, in altre parole i dispositivi Nokia, posseggono ancora una grossa fetta di mercato, nonostante la crescente diffusione di iPhone e Android.

Cosa serve per sviluppare:

- Qualsiasi PC con Windows, Linux o un Mac con OSX.
- Un IDE/compilatore Java (es. Eclipse) o C++ (Visual C++ oppure Visual Studio con C++)

Conoscenze richieste: C++ per "applicazioni native", oppure J2ME (mobile edition) per applicazioni eseguite dentro il runtime Java-based. Le applicazioni Java sono tipicamente più lente e limitate (nessun accesso a basso livello o funzioni avanzate come intercettazioni di eventi di sistema ecc.). C++ per contro permette di fare di tutto sul Symbian, personalizzando qualsiasi aspetto. Purtroppo lo sviluppo risulta essere comunque molto impegnativo, anche per chi conosce già il C++.

Sviluppare in Java ha anche il vantaggio che l'applicazione potrà poi essere portata anche su altre piattaforme con sforzi spesso molto ridotti, a volte addirittura assenti. Per chi viene dal mondo .NET, imparare J2ME è una questione di pochi giorni. Chi invece già conosce Java deve solo vedersi le poche classi per creare l'interfaccia utente. In generale è un compito tranquillamente alla portata di chiunque abbia già delle buone basi di programmazione object oriented moderna.

Pro:

- Enorme diffusione della piattaforma.
- Facilità di sviluppo con Java.
- Portabilità delle applicazioni sviluppate con Java.
- Costi di sviluppo molto ridotti o assenti.
- Libertà di scelta di vari canali di distribuzione (sul proprio sito, su store di terze parti ecc.).
- Il supporto per il mondo Java (sotto forma di forum, libri, corsi) non manca.

Contro:

- Esistono centinaia di diverse configurazioni hardware di cui tenere conto, e questo può facilmente diventare un problema per uno sviluppatore che deve testare il software in infiniti simulatori, differenziare parti di codice, scrivere istruzioni dettagliate per modelli diversi, compilare una lista di compatibilità ecc.

2.4.3 BlackBerry

I BlackBerry prodotti da RIM (Research In Motion) sono dispositivi relativamente nuovi nel mercato italiano, ma già da molti anni popolarissimi negli USA, dove RIM è tra le prime 3 aziende del settore. Storicamente associati esclusivamente al mercato business (manager e impiegati che hanno bisogno di essere sempre collegati alla mail aziendale), gli ultimi modelli sono diventati di interesse anche per la massa degli utenti consumer, attenti alle funzioni multimediali, all'estetica, alla ricchezza di applicazioni.

Cosa occorre per sviluppare:

- Qualsiasi PC con Windows, Linux o un Mac con OSX.
- Eclipse + un plug-in di RIM; oppure il JDE, un ambiente derivato da Eclipse fatto su misura per lo sviluppo con BlackBerry
- Un set di chiavi per firmare digitalmente le proprie applicazioni. Si comprano tramite il sito di RIM, e costano solo 20\$, una-tantum.

Conoscenze richieste: J2ME, come per Symbian. Ovviamente oltre alle classi generiche di J2ME RIM ha aggiunto tutta una serie di classi specializzate per accedere alle funzionalità e caratteristiche peculiari del proprio sistema operativo e dei propri dispositivi, ma tipicamente si tratta solo di classe che ereditano da una classe J2ME base e la estendono con nuove proprietà.

Pro:

- Semplicità di sviluppo.
- Non ci sono libri dedicati ma la documentazione ufficiale è piuttosto completa, gli esempi ufficiali coprono praticamente tutto, e il forum di supporto per sviluppatori è molto efficiente
- Esistono decine di device con caratteristiche diverse, ma sono tutti prodotti da RIM, quindi è più semplice gestire la compatibilità
- RIM fornisce i simulatori di tutti i device, in modo da semplificare di molto il test delle proprie applicazioni
- Tra le classi di J2ME e le estensioni di RIM è possibile interagire con il sistema operativo e le applicazioni di sistema in modo piuttosto completo: si possono intercettare molti eventi di sistema, scrivere delle applicazioni che girano in background e partono all'avvio, mandare SMS e mail, accedere al GPS, utilizzare connessioni verso indirizzi remoti, accedere ai dati del calendario e della rubrica, disegnare interfacce personalizzate, gestire suoni e video ecc.

Contro:

- SDK parzialmente limitato per alcuni aspetti: non è ad esempio possibile cancellare SMS, intercettare degli eventi particolari, cambiare alcune impostazioni di sistema (profilo attivo, audio, immagine di sfondo della schermata principale) o cambiare il comportamento di alcune applicazioni di sistema.
- I file prodotti dalla compilazione (.JAR e .COD) non possono superare una certa dimensione massima. Ad esempio un file di 8MB (dimensione raggiunta includendo al suo interno molti file di testo, immagini ecc.) non verrà installato nel telefono.

- Non è presente un DB Engine installato di default. SQL Anywhere 11 di Sybase include UltraLite J per BlackBerry, un motore di database relazionale che può andar bene in molte situazioni, ma è comunque un software da installare separatamente, e ha le sue limitazioni (soprattutto sul numero di oggetti gestibili)

2.4.4 Windows Mobile

Ci sono in commercio tantissimi dispositivi diversi, di produttori diversi. WM è un sistema operativo piuttosto stabile e dalle mille possibilità.

Cosa serve per sviluppare:

- Qualsiasi PC con Visual Studio 2005/2008 Standard o Professional

Conoscenze richieste: C#/VB.NET e il .NET Compact Framework. A rendere lo sviluppo più semplice c'è Visual Studio, considerato uno dei più avanzati IDE, con potenti strumenti di design visuale delle interfacce grafiche, un editor di codice e un debugger eccellenti, e tutta una serie di strumenti di contorno che semplificano di molto la vita.

Pro:

- Semplicità di sviluppo.
- Ricca disponibilità di libri, tutorial, screencast, esempi e documentazione di vario tipo. Come per lo sviluppo in altri ambiti, Microsoft si distingue positivamente in questa categoria.

- Disponibilità di SQL Server Mobile Edition per la gestione di complessi database relazionali direttamente sul device (con la possibilità di sincronizzarsi con SQL Server)
- Ampia libertà di sviluppo: l'SDK permette di fare praticamente di tutto. Dove non sia possibile fare qualcosa con il .NET Compact Framework, è di solito possibile farlo con C++ ad un livello più basso
- Nessuna approvazione e/o firma da organizzazione esterne è necessaria per il deployment dei propri applicativi custom
- Grande disponibilità di componenti commerciali (es: griglie avanzate o controlli per realizzare grafici) già pronti all'uso

Contro:

- Creare un'interfaccia utente accettabile per i propri programmi significa sostanzialmente crearsela da zero. Del resto così facendo si perde l'uniformità con il resto delle applicazioni - per quanto brutta la UI, di solito una qualche uniformità è utile.

2.4.5 Google Android

Alcuni la definiscono la piattaforma mobile del futuro, altri come un tentativo di raggiungere i risultati di Apple. I numeri della piattaforma di Google parlano però di 300 mila unità attivate ogni giorno, ed un'offerta di applicazioni (Android Market) in rapidissima crescita.

Cosa serve per sviluppare:

- Qualsiasi PC, Mac Intel-based o Linux
- Android SDK (gratuito)
- Eclipse / Netbeans IDE con gli opportuni Plug-in

- Una quota di 25\$ per l'iscrizione nel *Market*

Conoscenze richieste: Si tratta di uno stack software basato sul kernel Linux, ma le applicazioni sono eseguite da una Virtual Machine Java. Il meccanismo a supporto del layout grafico è gestito principalmente da documenti XML. Android supporta, a livello nativo, le librerie SQLite, OpenGL ed uno stack SIP.

Pro:

- Sistema Operativo completamente *Open Source*
- Viene dato pieno accesso alle API di sistema: controllo di tutti i componenti del dispositivo
- Se qualche funzionalità non è inclusa nelle API ufficiali, o è considerata di basso livello, è sufficiente dare uno sguardo al codice sorgente della piattaforma
- Qualsiasi applicazione può intercettare e gestire un qualsiasi evento di sistema
- Qualsiasi applicazione può essere eseguita in background inoltre è possibile creare dei servizi con le stesse priorità di quelli di sistema
- Viene data la possibilità agli sviluppatori di gestire l'esecuzione delle applicazioni in processi/thread diversi
- L'architettura del framework è fortemente orientata all'interoperabilità delle applicazioni
- Il fatto che le applicazioni siano scritte in Java garantisce il supporto della più grossa comunità di sviluppatori esistente

Contro:

- L'elevata varietà di dispositivi può rendere difficile lo sviluppo di applicazioni che fanno affidamento sulle prestazioni e peculiarità hardware.

2.4.6 La Piattaforma scelta: Android

Non è chiaro quale piattaforma diventerà uno standard per gli sviluppatori per la telefonia mobile, e se ci sarà una piattaforma che dominerà il mercato. Gli analisti differiscono l'uno dall'altro nelle loro valutazioni sulle quote di mercato attuali e future. I dati parlano chiaro riguardo alla posizione di Nokia e Symbian, una volta dominante, che sta subendo una forte erosione da parte di RIM, Apple e Google.

Per scegliere la piattaforma più adatta per il nostro progetto potremmo quindi valutare la convenienza dal punto di vista dello sviluppo: Android è la piattaforma più “aperta” (rispetto a Symbian, unica altra Open Source), supporta una facile integrazione con Eclipse, ed usa una “variante” di Java.

Inoltre potremmo valutare quale sistema si adatta meglio al modello Context-Aware. Prendiamo in considerazione la struttura generale della maggior parte dei middleware costruiti a strati. La stratificazione impedisce alle applicazioni ad un livello più alto di venire a conoscenza degli eventi riguardanti i livelli inferiori. In Android non esiste questa barriera. Ogni applicazione può essere al corrente dello stato del sistema e degli eventi che lo riguardano, fornendo di fatto il pieno supporto per il Context-Aware. Questo è reso possibile dal meccanismo di notifica degli eventi noto con *Intent Delivery Framework*. Inoltre tutta la piattaforma è intesa come un insieme di componenti “standalone” che possono essere connessi al sistema quando il contesto lo richiede.

Queste considerazioni portano alla luce qualcosa in più di un'ennesima piattaforma basata su Java, infatti la sua architettura si colloca diversi passi avanti rispetto alle piattaforme concorrenti.

2.5 Android ed il *Context-Aware*

Gli stessi motivi che hanno guidato la nostra scelta hanno portato, molto probabilmente, alla medesima decisione da parte di due università.

Al corso di Laurea di *Computer Science* della “George Mason University” vi è un esame che si propone di impartire agli studenti le nozioni per progettare e sviluppare *Software for Context-aware Multi-user Systems*.

Mentre alla “University of Bristol” vi è un corso simile chiamato *Mobile and Ubiquitous Computing* che richiede invece, come esame finale, la progettazione di un sistema Context-Aware.

Ecco inoltre un breve elenco di alcune applicazioni presenti sul Market di Android.



IST_MUSIC

Si tratta di una framework “open” per lo sviluppo di applicazioni auto-adattanti in base al contest. Include un “runtime environment”, strumenti di sviluppo e linee guida.



CAB

Il CAB (Context-Aware Browser) è un browser che presenta all’utente pagine ed applicazioni web in relazione al contesto corrente. Il contesto indicato è però strettamente correlato all’evento “Giornate del Tiepolo” a Udine, svoltosi dal 21 al 30 maggio 2010.



Tracklet

Applicazione finalista della gara *ADC2* indetta da Google stessa. Tracklet si propone come strumento per facilitare il “budgeting” , o pianificazione delle spese. Ha potenzialità “time and location aware”, in altre parole le spese e fatte sono legate alla data ed il luogo in cui sono state effettuate.



Workaholic

Permette di tracciare le ore di lavoro effettuare calcolando automaticamente l’orario di entrata e di uscita dal posto di lavoro. Utilizza ovviamente funzionalità location-aware.



Cab4Me

Utilizzando la propria posizione come dato di contesto, permette di rintracciare facilmente i taxi a noi più vicini. Segnala i punti taxi e le compagnie più vicine fornendo inoltre i numeri per contattarli ed altre informazioni utili, come il metodo di pagamento accettato.



BreadCrumbz

Letteralmente “Briciole di Pane” è un’applicazione che aiuta l’utente, con un processo guidato, a creare il proprio percorso da un punto A ad un punto B. Si possono inserire punti di interesse e si può continuare i percorsi anche all’interno degli edifici.

3. Android

Google non è solo il nome che si associa al celeberrimo motore di ricerca, ma è una fonte inesauribile di idee specialmente in ambito Web2.0. E' sufficiente andare all'indirizzo labs.google.com per accorgersi di tutti gli strumenti che questa azienda mette a disposizione quali API per la realizzazione di applicazioni basate, ad esempio, sull'utilizzo delle Google Map o sugli stessi strumenti di ricerca del famoso motore.

3.1 Introduzione ad Android

Android è un sistema operativo open source basato sul kernel 2.6 di Linux; la prima idea nasce a Palo Alto, California, da una piccola azienda di nome Android Inc. fondata da Andy Rubin (a sua volta co-fondatore di Danger), Rich Miner (co-fondatore di Wildfire Communications, Inc.), Nick Sears (vice presidente di T-Mobile), e Chris White (principale autore dell'interfaccia grafica di WebTV).

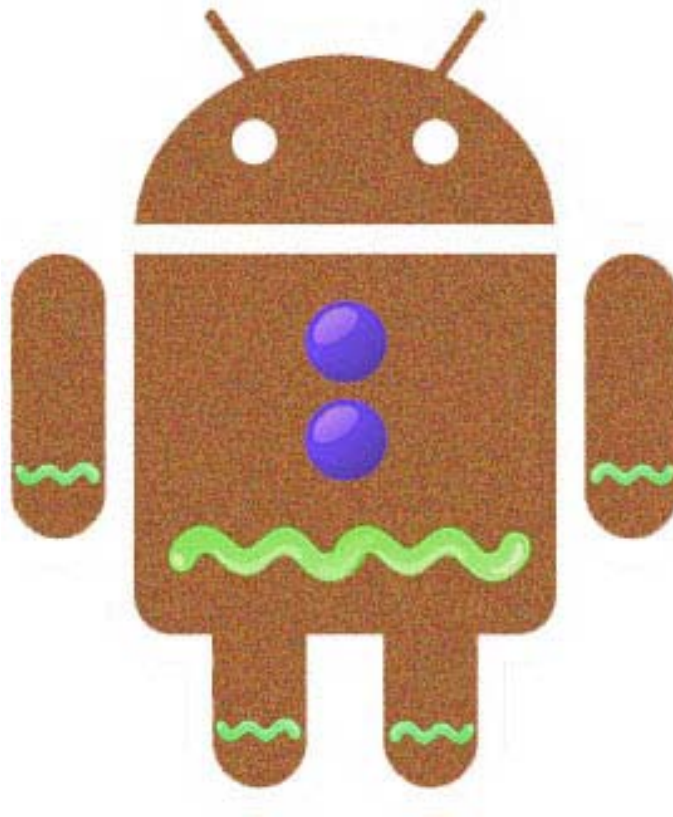
Nel Luglio 2005 ci fu la vera e propria svolta, Google acquista Android Inc. e i suoi fondatori trovano le risorse necessarie per incrementare lo sviluppo del sistema operativo anche grazie all'OHA (Open Handset Alliance) un accordo tra svariate compagnie, con a capo Google, il cui obiettivo è creare e sviluppare "standard aperti" per dispositivi mobili.

Il 5 Novembre 2007 OHA viene istituita ufficialmente e nello stesso giorno presenta Android. Pochi giorni dopo esce il primo Software Development Kit (SDK) alla versione 1.0_r1 e quasi un anno dopo (23 Settembre 2008) il primo

cellulare con Android, il T-Mobile G1; da lì in poi la galoppata di Android è stata inarrestabile, sono arrivati in un solo anno cinque aggiornamenti:

- **1.1** uscito il 9 Febbraio 2009 manteneva la retrocompatibilità con i suoi predecessori e aggiungeva nuove API (Application Programming Interface);
- **1.5 Cupcake** uscito il 30 Aprile 2009 con importanti novità, tra cui la possibilità di scattare foto, registrare video e caricarli direttamente su Picasa o YouTube, una nuova tastiera a video e le animazioni tra gli schermi;
- **1.6 Donut** uscito il 15 Settembre 2009 con le prime apparizioni di Voice Search e il navigatore Turn-by-turn oltre a nuove interfacce grafiche per la galleria fotografica e il Market;
- **2.0 / 2.1 Eclair** uscito il 26 Ottobre 2009 è la versione più conosciuta e utilizzata di Android. Supporto all'HTML5 e al flash per la fotocamera, zoom digitale, live wallpapers, incremento della tastiera a schermo e supporto a differenti misure e risoluzioni per gli schermi oltre a varie nuove applicazioni come Google Maps 3.1.2;
- **2.2 Froyo** uscito il 20 Maggio 2010. Con possibilità di tethering USB e Wi-Fi (poter trasformare il proprio smartphone in un *hotspot*), supporto a Adobe Flash 10.1, supporto all'installazione su micro sd, miglioramento della velocità (da 2 a 5 volte più veloce), memoria e performance del sistema operativo, aggiornamenti delle apps del Market con la possibilità di aggiornamento automatico.
- **2.3 Gingerbread** uscito il 6 dicembre 2010. Migliorate le performance della Dalvik Virtual Machine, migliorate le performance del motore grafico, ottimizzate le funzionalità di scrittura, aggiunto uno stack SIP per il supporto alla comunicazione VOIP.

Figura 5 Il logo di Android 2.3



La caratteristica principale che differenzia Android da altri sistemi operativi è la sua vocazione open source: sotto l'acronimo AOSP (Android Open Source Project) si sviluppa il progetto Android. Il codice sorgente di tutti i livelli del telefono (dal kernel, alle librerie, fino al livello applicativo) è scaricabile da internet e modificabile a proprio piacimento. Così facendo OHA si è guadagnata il favore e il rispetto di una nutrita schiera di programmatori che hanno potuto non solo sviluppare applicazioni per il proprio cellulare (anche altri sistemi distribuiscono SDK per fare ciò), ma anche di dare una sbirciata al funzionamento dei livelli inferiori e contribuire all'evoluzione della piattaforma con *bugs fix*, suggerimenti e proposte di codice.

I dati sulla diffusione della piattaforma sono alquanto sbalorditivi: si è passati dal 2,8% del secondo quadrimestre 2009 al 13% di luglio 2010. Inoltre la contestuale crescita del mercato Apple indica che Android sta assorbendo una

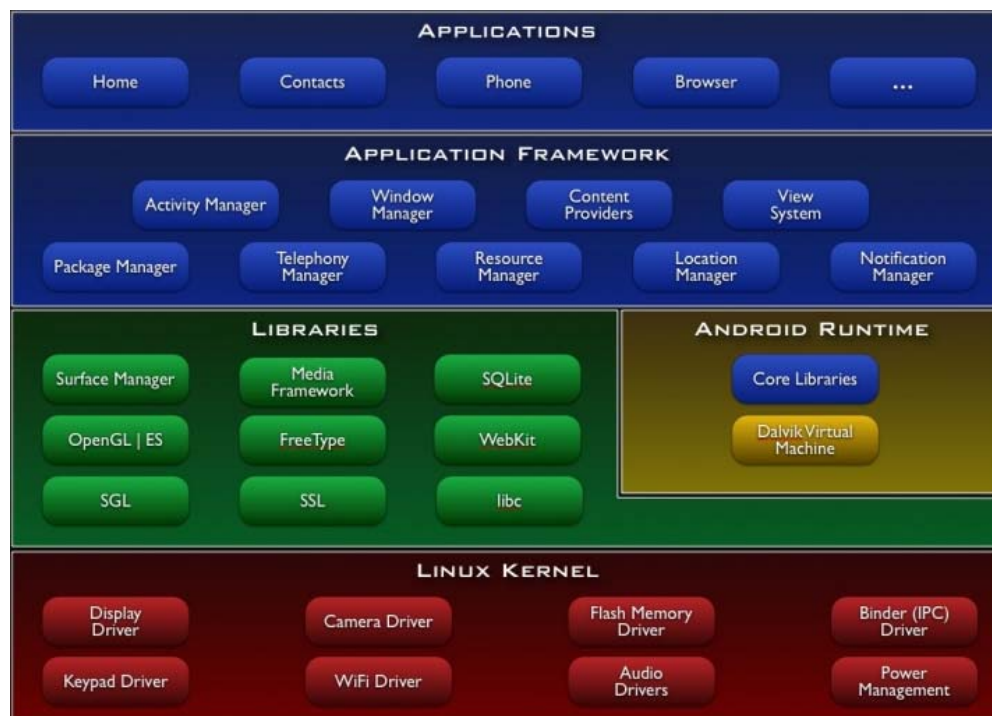
fetta di mercato del Blackberry. Gli analisti ritengono che questo rapido successo, molto simile a quello di Microsoft Windows Mobile, sia dovuto alla natura variegata dei produttori che supportano la piattaforma Android.

3.1.1 Architettura

Android è un *software stack* per mobile devices che comprende al suo interno sistema operativo, middleware e un'insieme di *key applications*. Sono 5 i layers che collaborano tra loro per la sua realizzazione: si parte ovviamente dal livello più basso che si occupa della gestione fisica del dispositivo, fino ad arrivare al livello applicativo. Tra questi due estremi troviamo layers che permettono il funzionamento logico e applicativo dello smartphone.

Nella figura in basso tutti i componenti in azzurro sono scritti in Java.

Figura 6 Architettura Stack Android



Linux Kernel: Si tratta dello Standard Kernel Linux 2.6.24 con delle patch per supportare Android. Non possiede un sistema di gestione a finestre (come *Xwindow*), non ha il supporto per *glibc*, e soprattutto non include tutto il set di comandi standard di Linux. I motivi per cui viene utilizzato è che offre un'ottima gestione della memoria e dei processi, driver efficienti e testati, sistema di sicurezza a permessi, supporto per le librerie condivise e inoltre è open source.

Libraries: Sono un insieme di librerie necessarie al funzionamento del sistema. E' possibile dividerle qualitativamente in quattro gruppi distinti per tipologia di servizi offerti:

1. *Bionic*: implementazione ottimizzata per soluzioni embedded delle librerie standard di C. Viene utilizzata questa soluzione in quanto è necessario limitare al minimo il consumo di risorse da parte di un processo. Ovviamente è stato necessario effettuare una scelta ben precisa tra operazioni che si volevano o meno supportare.
2. *Function Libraries*: Insieme di librerie per la gestione del browser con supporto per CSS, javascript, AJAX, DOM (WebKit), per il supporto codec audio/video (Media Framework) e per un datastore transazionale a basso costo in termini di risorse (SQLite).
3. *Native Servers*: Ci sono server per diversi scopi: tra i più utilizzati quelli per la gestione del rendering e della composizione delle interfacce dell'applicazioni sullo schermo (Surface Flinger) e quello per la gestione dei device audio (Audio Flinger).
4. *Hardware Abstraction Libraries*: Scritte in C/C++ definiscono le interfacce che i driver per l'hardware di Android devono implementare. Il loro scopo è quello di separare la parte della logica del sistema dalle interfacce hardware. Queste vengono realizzate nello user-space in quanto è possibile che non esistano kernel driver standardizzati per tutti i componenti e Android da questo punto di vista ha requisiti ben specifici. Ovviamente queste librerie vengono caricate a run-time in caso di necessità.

Android Runtime: Si tratta del layer nel quale vengono implementate le Virtual Machine: il suo compito è quello di permettere la portabilità delle applicazioni e la consistenza a runtime. Per funzionare in maniera più efficiente utilizza un particolare formato di file (.dex) e il *Dalvik bytecode*. I file Java .class/.jar vengono convertiti nel Dalvik bytecode nella fase di compilazione.

La **Dalvik Virtual Machine** è specificatamente realizzata per embedded environment: utilizza sapientemente la memoria, supporta più VM per device e il bytecode interpreter è molto efficiente, inoltre dall'ultima versione è stato introdotto il compilatore *Just In Time*.

Le **Core Libraries** sono API Java che provvedono alla realizzazione di una semplice ma potente *development platform* (data structures, utilities, file access, network access, graphics....)

Application Framework: Sono tutti quei componenti essenziali per il funzionamento del sistema che le applicazioni non accedono direttamente: Activity Manager, Package Manager, Window Manager, Resource Manager, Content Provider, View System.

Vi sono poi un'insieme di servizi detti anche **Hardware Services** il cui compito è fornire delle API per garantire un accesso al livello fisico del dispositivo. Il sistema è realizzato in questo modo per controllare il comportamento delle applicazioni e fare in modo che queste non abbiano una visione diretta dell'hardware.

Application: si tratta del livello più alto dello stack di Android. E' il layer in cui risiedono le applicazioni.

3.1.2 Componenti Fondamentali

Il linguaggio utilizzato per scrivere le applicazioni in Android è Java. Un tool, chiamato *aapt*, ha il compito di prendere il codice e tutti i file di dati e risorse, compilarlo e generare un *Android Package*. Si tratta di un particolare archivio identificato dal suffisso *.apk*. Questa tipologia di file è utilizzata per la distribuzione degli applicativi: tutto il codice in un singolo *.apk* è considerato facente parte di un'applicazione.

- Ogni applicazione di Android vive in un proprio mondo:
- Viene eseguita in un proprio processo Linux.
- Ogni processo ha una propria Java Virtual Machine (JVM) in modo tale che è valida la proprietà dell'*isolation*: ogni processo esegue separatamente dal codice degli altri processi.
- Ogni applicazione è associata ad un unico Linux User ID. (E' possibile realizzare ed utilizzare una modalità condivisa in cui più applicazioni condividono User ID, JVM e processo Linux).

Una delle caratteristiche più interessanti di Android è che un'applicazione può utilizzare gli elementi di un'altra, senza incorporarne il codice, semplicemente invocando l'attività dell'altro programma che intende sfruttare. Per offrire questa funzionalità il sistema deve essere in grado di avviare autonomamente un'application process nel momento in cui questo è richiesto. Le applicazioni di Android, al contrario di altri sistemi, non hanno un singolo *entry point* (Es. *main()*), ma hanno componenti che possono essere istanziati dal sistema nel momento del bisogno:

Activities: Parte dell'applicazione che presenta una *Visual User Interface* che viene visualizzata sullo schermo e attraverso la quale un utente interagisce. Tipicamente una delle attività dell'applicazione viene settata come principale ed è quella che viene mostrata all'utente nel momento in cui l'applicazione viene lanciata. Cliccare o aprire differenti viste corrisponde al muoversi da un'attività all'altra. Il

contenuto della finestra è composto da una gerarchia di *View Component*: sono componenti che fungono da contenitori e permettono di inserire all'interno altri elementi. I contenitori (detti anche *Parent View*) sono responsabili della rappresentazione degli elementi contenuti (*Leaf Component*), i quali a loro volta devono gestire l'interazione dell'utente con la zona da loro occupata.

Services: un servizio non ha un'interfaccia grafica, ma lavora in background per un tempo indefinito. Ad esempio potrebbe dover di recuperare delle informazioni dalla rete, oppure riprodurre della musica. Quest'ultimo caso è molto esplicativo per comprendere la differenza tra un Servizio e un Attività. Poniamo di avere un Music Player, un'applicazione che riproduce file di tipo audio. Se volessimo continuare ad ascoltare musica mentre sfogliamo la rubrica, non potremmo sviluppare il lettore collegato solamente all'attività Music Player: nel momento in cui essa perde il focus e viene rimpiazzata da un'altra attività il lettore smetterebbe ovviamente di riprodurre. Dunque è necessario implementare un apposito servizio che rimanga attivo in background e continui a riprodurre i files musicali. Ovviamente l'accesso e la gestione del servizio dovrà avvenire tramite l'attività Music Player: ma ciò non ci vincolerà comunque a mantenere il focus su questa attività.

Broadcast Receivers: si tratta di un componente che ha il compito di ricevere e reagire ad annunci di tipo *broadcast*. All'interno del sistema possono essere generati diversi segnali broadcast (cambio fuso orario, livello batteria basso, cambio preferenza lingua...). I Broadcast Receivers non presentano alcuna interfaccia grafica, ma al contrario possono fare in modo che ne venga attivata una o possono avvisare l'utente dell'arrivo di un messaggio tramite il Notification Manager.

Content Provider: Il compito di un content provider è quello di fare in modo che i dati di un'applicazione siano disponibili a chi ne può avere bisogno. I dati possono essere salvati in diversi modi (SQLite DB, file system, etc...) e la scelta è ovviamente a discrezione del programmatore. Un provider, per essere accessibile alle altre applicazioni, deve implementare l'interfaccia

contentProvider. L'accesso ad un Content Provider non avviene mai direttamente ma tramite un *ContentResolver*: passando l'URI della risorsa richiesta il content resolver si incarica di avvisare il sistema della necessità di un certo dato. Il sistema avvierà lo specifico ContentProvider che fornirà l'accesso a quella risorsa.

3.1.3 Interazione tra Componenti

Tutti i componenti (tranne i Content Provider) sono attivati da messaggi asincroni denominati *Intents*. Semplificando si può affermare che un Intent è un oggetto che contiene un messaggio. Per le Activities e Services, tra le altre cose, identifica l'azione richiesta e specifica l'URI dei dati su cui agire. Oppure, nel caso dei Broadcast Receiver, un oggetto Intent identifica l'azione per cui il Broadcast Receiver è "in ascolto".

Per inizializzare ogni componente vi sono metodi differenti: per le attività si passa come parametro un Intent object specifico alla funzione *Context.startActivity()* o a *Activity.startActivityForResult()*. Per un servizio il funzionamento sarà corrispondente (si passa l'Intent a *Context.startService()*). Similmente per i broadcast receiver avremo la funzione *Context.sendBroadcast()* che si fa carico di consegnare l'Intent ai listener interessati.

A differenza degli altri componenti, i ContentProvider sono attivati quando sono oggetto di una richiesta da parte di un *ContentResolver*.

Affinchè Android possa utilizzare un componente di un'applicazione deve sapere che questo esiste. E' dunque necessario dichiararlo all'interno di un particolare file xml (*AndroidManifest.xml*) contenuto assieme ai dati e al codice in tutti i pacchetti *.apk*. Lo scopo di questo file è quello di informare Android dei contenuti dell'apk stesso. All'interno, oltre ad elencare i componenti e alcune informazioni circa l'applicazione stessa, vi sono indicate le librerie utilizzate e i

permessi che l'applicativo necessita per funzionare correttamente. Per quanto riguarda la sintassi completa del file (come dichiarare attività, servizi, receiver, provider, etc) si rimanda alla documentazione ufficiale. Si ritiene però interessante soffermarsi su una tipologia di tag che permette di comprendere meglio il funzionamento del sistema: *<intent filter>*

Nell'Intent viene esplicitamente indicato un target a cui deve essere inviata la comunicazione. Android troverà quel componente all'interno del sistema (basandosi ovviamente sulle informazioni ricavate da tutti i manifest files) e lo attiverà passandogli tutti i parametri addizionali. Ma se questa specifico componente non è localizzabile, Android intelligentemente cercherà l'applicazione migliore da attivare in grado di gestire ugualmente la richiesta. Questa operazione di ricerca verrà fatta sulla base delle informazioni contenute alla voce *<intent filter>* del manifest file di ogni applicazione installata.

Una delle principali caratteristiche di Android è che una attività può avviare un'altra, anche se quest'ultima appartiene a un'applicazione diversa. Ciò possibile per il semplice fatto che la struttura degli applicativi, non considerando la complessità e la dimensione, è sempre la medesima. Inoltre, come già descritto precedentemente, il sistema è sempre al corrente (tramite i file manifest) di tutti i componenti presenti e attivabili.

Ipotizziamo di voler creare un'applicazione che permetta di cercare dei ristoranti. L'obiettivo sarà quindi mostrare all'utente le recensioni e l'ubicazione dei ristoranti di una certa zona. Sarebbe comodo poter visualizzare direttamente su una cartina (Es. Google Maps) e non solamente in formato testuale la posizione del ristorante per valutarne meglio la distanza. Nel caso si avesse già a disposizione un'applicazione che, dato un certo indirizzo, lo visualizzasse su una mappa, si potrebbe utilizzare direttamente all'interno del sistema per la localizzazione dei ristoranti senza dover scrivere un modulo apposta. All'utente ciò apparirà come se la mappa sia parte integrante dell'applicazione di ricerca dei ristoranti anche se in verità essa è definita in un'altra applicazione ed è eseguita nel proprio application's process.

La politica di Android per la gestione dei componenti consiste nel mantenere le due applicazioni nel medesimo *Task*. Con *Task* si intende la cosiddetta *User Experience* ovvero il gruppo di attività attualmente in uso ordinate a stack. Si denomina *root activity* del task quella che lo ha inizializzato e che tipicamente è un'applicazione che l'utente seleziona nell'application launcher. L'attività al top dello stack è quella che è al momento funzionante e che ha il focus per la gestione dell'user actions. Quando viene inizializzata un'altra attività, quest'ultima viene posta al top dello stack e quella precedentemente attiva occuperà la seconda posizione. Nel momento in cui viene premuto il tasto BACK l'attività precedente viene recuperata, riavviata e riposizionata in cima allo stack.

3.1.4 Processi e Thread

Quando il primo componente di un'applicazione deve essere lanciato, Android inizializza un processo Linux con un singolo Thread. La politica di default è che tutti i componenti dell'applicazione verranno eseguiti in quel thread di quel processo. Ovviamente è possibile modificare discrezionalmente questa politica per realizzare dinamiche multi-threads e/o multi-processes.

Processes: Il processo in cui un componente viene eseguito è impostato all'interno manifest file. Ogni elemento (<activity>, <service>, <receiver>, <provider>) ha a disposizione un attributo (*process*) che specifica in che processo debba essere avviato e ciò permette a componenti di applicazioni diverse di eseguire all'interno del medesimo flusso di esecuzione. L'elemento <application> (ovvero quello di massimo grado e che contiene tutti gli altri) ha anch'esso questo attributo che ha valore di default per tutti i componenti nei quali non è specificatamente definito. Poiché l'implementazione standard prevede che tutti utilizzano il medesimo thread all'interno dell'unico processo, è sconsigliato definire azioni bloccanti (per esempio dei listener in attesa di un certo carattere della tastiera) o particolarmente lunghe. La soluzione migliore consiste ovviamente nella creazione di un nuovo thread a cui adibirne lo svolgimento. Può succedere (e in uno smartphone ciò si verifica spesso) che Android necessiti di

terminare dei processi se un altro richiede spazio di memoria e quella attualmente a disposizione non sia sufficiente. Se l'applicazione *killed* sarà nuovamente richiesta il sistema dovrà farla ripartire recuperando le informazioni in uso precedentemente. Le politiche di scheduling, per scegliere quali processi terminare e quali lasciare in vita, sono realizzate tenendo conto dell'importanza del processo per l'utente. Ovviamente Android cercherà di mantenere in memoria più a lungo i processi che potrebbero essere riutilizzati.

Threads: Anche se ciascuna applicazione ha un proprio processo ci possono essere casi in cui è necessario dover usufruire di threads per effettuare delle operazioni in background. Per esempio visto che la user interface deve rispondere rapidamente alle azioni dell'utente, il thread che gestisce questa interazione non può sospendersi in operazioni di network activities. Pertanto tutto ciò che non riesce a completare rapidamente dovrebbe essere assegnato ad un altro thread.

Remote Procedure Calls: Android mette a disposizione un *lightweight mechanism* per eseguire RPC. Una RPC permette, tramite una chiamata locale, l'esecuzione di un metodo remoto a carico di un altro processo. Il sistema garantisce un completo supporto alla realizzazione e la gestione di questa tipologia di operazioni. Il programmatore avrà solamente l'onere di definire e implementare le interfacce. I metodi così definiti vengono eseguiti in modalità *synchronized* (solo un metodo per volta) anche se non hanno un return value. La realizzazione del servizio non è molto differente da una normale implementazione di una RPC: vengono dapprima scritte le interfacce usando IDL (Interface Definition Language) e tramite un tool si ottengono le classi di supporto. Ovviamente il programmatore dovrà implementare il client (colui che richiederà il servizio e si porrà in attesa del risultato) e il server (colui che fornirà il servizio e produrrà il risultato).

Thread-safe methods: In alcuni casi un metodo invocabile da remoto, può essere utilizzato da più threads. E' necessario pertanto che sia realizzato in modalità thread-safe. Ovvero si fa in modo che le sezioni critiche vengano accedute solo da un thread per volta. Questo è di particolare importanza per

quanto riguarda le RPC e l'accesso alle risorse tramite ContentProvider. In quest'ultimo caso dobbiamo evitare che due threads concorrenti accedano contemporaneamente alle risorse e le modifichino in maniera incontrollata.

3.2 Android vs iPhone

Il 2007 è l'anno di svolta per il settore telefonico, in particolare per il segmento degli smartphone.

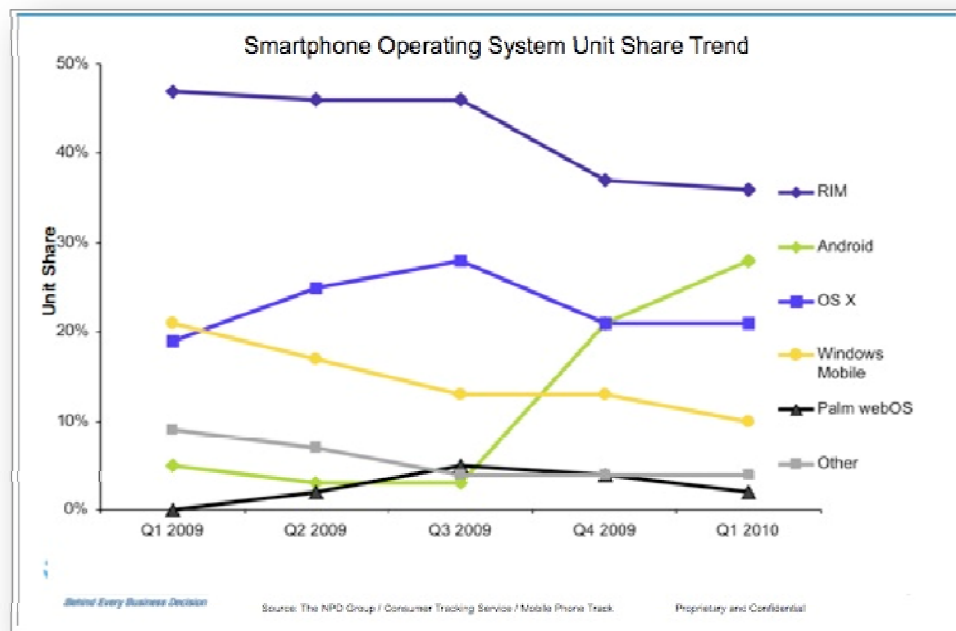
Il 9 gennaio viene presentato *iPhone* ed a luglio viene immessa nel mercato la prima generazione dello telefono Apple. Grazie alla forza trainante del suo brand essa è riuscita ad avvicinare il grande pubblico al settore degli smartphone. Non lo ha fatto proponendolo direttamente come una piattaforma applicativa, ma bensì prima si è fatto apprezzare per il proprio design e solo dopo l'utenza media ne ha compreso e incominciato a sfruttare le potenzialità. Da quel momento il mercato ha subito una forte deviazione: infatti mentre qualche anno fa erano pochissimi i modelli di smartphone disponibili, ora tutte le case produttrici hanno una propria famiglia di prodotti dedicata a questo segmento.

Figura 7 I Loghi Apple e Android



Fare un confronto diretto tra Android e iPhone è davvero difficile vista la diversa natura dei contendenti. L'ovvia motivazione risiede nei dispositivi hardware coinvolti. Quando si parla di iPhone ci si trova davanti al sistema operativo (iOS) e due soli dispositivi disponibili. Mentre quando si parla di Android ci si riferisce al solo sistema operativo ma non si ha nessun dispositivo di riferimento in particolare. Differenza da tenere in considerazione quando di guarda ai risultati di vendita dei dispositivi. Nel grafico seguente infatti possiamo notare il sorpasso di Google nel primo quadrimestre 2010: parte di questo sorpasso è avvenuto grazie alla tendenza, di alcuni produttori, di vendere smartphone Android anche con hardware di fascia medio/bassa (Graham, 2010).

Figura 8 Grafico Vendite Smartphone



Tante le caratteristiche divergenti, ovviamente alcune sono più importanti di altre. Come l'accesso gratuito al mondo degli sviluppatori Android, a differenza dei 99 dollari necessari per la sola iscrizione al "Programma Sviluppatori iPhone".

Entrambi offrono funzionalità di *multitasking* ma, ancora una volta, lo fanno in modo diverso. Quello di Android è del tutto simile a quello di un qualsiasi PC: passando ad una nuova applicazione quella precedente continuerà ad essere eseguita in secondo piano (*background*). Ovviamente questo comporta anche consumo di batteria, memoria e tempo del processore. Per questo motivo il sistema gestisce automaticamente il “sovraffollamento” delle applicazioni ed offre strumenti di analisi per controllare come vengono usate le risorse (Frank, 2010).

Il multitasking fornito da iOS è disponibile solo per le applicazioni create da Apple. Le applicazioni di terze parti hanno a disposizione funzionalità limitate: possono eseguire solo piccole parti dell'applicazione.

La sfida si fa sempre più avvincente con l'avvicinarsi dei vari aggiornamenti: test sulle prestazioni web ed esecuzione Javascript danno vincente la penultima versione di Android (2.2 “Froyo”) contro l'ultima di Apple (iOS4).

Ma la vera “guerra” è quella sulle applicazioni, o “Apps”, disponibili nei rispettivi market. Ad oggi il divario tra le oltre 100 mila di Android e le quasi 300 mila di iPhone, di cui occorre tener conto della maggiore anzianità nel mercato, è ancora alto. La differenza maggiore è anche nel tipo di “market”: quello di Google rimane fedele alla concezione di aperto e libero, quello di Apple ha un approccio davvero molto *Orwelliano*.

L'uomo della mela (dopo Sir Isaac Newton ovviamente) Steve Jobs, nel corso di una conferenza stampa dedicata agli analisti, ha dichiarato che la natura *Open Source* di Android è una debolezza e non un punto di forza. Secondo Jobs, Google ha creato un sistema troppo frammentato, con produttori smartphone che realizzano decine e decine di versioni personalizzate del sistema operativo. Le conseguenze sarebbero a carico degli sviluppatori, che si ritrovano a programmare software con il fardello di dover garantire compatibilità con tantissime versioni.

L'esempio presentato, secondo molti, è più che altro una drammatizzazione, perché in generale questo inconveniente si presenta quando si vuole integrare profondamente un software all'interno del Sistema.

La risposta a questa dichiarazione non si è fatta attendere troppo. Pochissime ore dopo uno dei creatori di Android, Andy Rubin, ha ribadito il concetto di *Open* scrivendo le 5 righe di codice necessarie per accedere al codice sorgente del Sistema Operativo. Il significato di quelle righe sta nella facilità con cui qualsiasi sviluppatore possa contribuire a migliorare il sistema operativo.

A prescindere da queste considerazioni molti analisti credono che Android potrebbe prendere il sopravvento e conquistare il mercato. Un'azienda specializzata in ricerche sul marketing prevede che otterrà il 30 per cento del mercato mondiale della telefonia mobile, mentre Apple solo il 15 per cento.

Il co-fondatore di Apple Steve Wozniak, intervistato da una rivista dei Paesi Bassi, ha ammesso che col tempo il sistema operativo di Google potrebbe assumere una posizione dominante nel mercato. Poco tempo dopo è arrivata una sorta di smentita. Ha dichiarato di non aver mai detto che Android fosse migliore di iOS, e che intendeva fare un parallelo con successo di *Windows* nel mercato dei sistemi operativi, sottolineando però come quest'ultimo non sia stato sempre di qualità nel corso degli anni. Nel corso dell'intervista ha parlato di come la qualità dell'iPhone sia un punto di forza difficile da superare, ma che i telefoni Android hanno più caratteristiche a disposizione e danno più possibilità di scelta per quelli che non vogliono essere limitati all'acquisto di un unico tipo di dispositivo (CNET, 2010).

3.3 Android ed il *Semantic Web*

Internet è come un organismo vivente. La prima versione (chiamata Web 1.0) offriva solo informazioni, ma con l'arrivo di social network, come *Facebook*, e imponenti siti di web commerce, come *Amazon*, la rete è diventata sempre più interattiva.

Con il Web 2.0 i contenuti sono prodotti principalmente dagli utenti, senza il loro contributo non esisterebbe Facebook, Wikipedia o Youtube. Col passare del tempo gli utenti hanno acquisito familiarità con queste potenzialità del Web 2.0, con attività di *blogging*, di *social networking* e di *social bookmarking* spianando la strada per il passo successivo nell'evoluzione di Internet. Un passo verso lo sviluppo del Web 3.0 "Intelligente" e "Onnipresente".

Si pensi ad esempio al "recommended system" che permette un approccio personale ad un sito web. Amazon utilizza questo sistema da tempo, potendo consigliare (recommend) ad un proprio cliente i prodotti acquistati da clienti precedenti aventi gli stessi interessi. Un altro esempio è il servizio offerto da *Last.fm* dove è possibile ascoltare musica online in relazione ai propri gusti musicali.

Il fattore comune tra i servizi descritti nei precedenti esempi la disponibilità di informazioni aggiuntive, relative ai prodotti offerti, fornite dagli utenti stessi.

Non ci sarà quindi un cambio istantaneo del Web ma una modifica graduale. Quello che cambierà in fretta invece sarà il "look & feel".

In parole povere Internet è una rete piena di computer che comunicano tra loro scambiando informazioni. Visto l'aumento di dispositivi connessi, il Web sarà sempre più presente ma meno visibile. Non sarà relegato all'uso da parte del solo PC ma sarà presente in modo invisibile nei dispositivi di tutti i giorni. Inoltre, il fatto che questi dispositivi comunichino tra loro attraverso Internet potrebbe portare a servizi aggiuntivi atti a soddisfare le esigenze individuali degli utenti.

Diventa quindi di importanza cruciale riuscire ad aggiungere informazioni a tutti dati presenti sul Web in modo da aggiungere semantica a semplici dati.

3.3.1 Il Semantic Web

Con Web semantico si intende la trasformazione del *World Wide Web* in un ambiente dove è possibile pubblicare non più solo documenti (pagine HTML, file, immagini, e così via) ma anche informazioni e dati relativi ai documenti stessi (metadati) in un formato adatto all'interrogazione, interpretazione e, più in generale, all'elaborazione automatica.

Così facendo sarà possibile un'interpretazione più significativa del contenuto dei documenti stessi, l'effettuazione di ricerche altamente precise, ed altre operazioni specialistiche.

L'evoluzione del web in web semantico inizia con la definizione, da parte del W3C, dello standard Resource Description Framework (RDF), una particolare applicazione XML che standardizza la definizione di relazioni tra informazioni ispirandosi ai principi della logica dei predicati (o logica predicativa del primo ordine) e ricorrendo agli strumenti tipici del Web (ad es. URI) e dell'XML (*namespace*).

L'utilizzo di questa nuova tecnologia sta progredendo con il tempo anche con notevoli sforzi istituzionali infatti la commissione europea sta finanziando ben 24 progetti relativi al Web semantico a partire dal 2006.

3.3.2 Semantic Reasoner

Un "Semantic Reasoner" è un software in grado di dedurre conseguenze logiche da un insieme di affermazioni di fatti o assiomi. La nozione di Semantic Reasoner generalizza quella di "Inference Engine" ("Motore Deduttivo", meccanismi di ragionamento automatico tipici dell'intelligenza artificiale)

fornendo un maggior numero di meccanismi con cui funzionare. Le regole di deduzione sono generalmente specificate attraverso linguaggi per la definizione di ontologie (come RDF).

Attualmente esistono diversi Reasoner in commercio, ma quelli più famosi tra i software open-source sono **Jena** e **Sesame**.

Sono entrambi dei framework open-source Java per sviluppare applicazione per il Semantic Web. Forniscono un ambiente di sviluppo per creare, manipolare ed interrogare basi di dati sematiche. Jena è un progetto nato dal programma per il Web Semanticode “HP Labs”, mentre Sesame è un progetto de “Aduna Open Source project”.

3.3.3 Androjena

Il concetto di Web Semantico è relativamente giovane, è stato presentato nel 2001 da Tim Berners-Lee, James Hendler e Ora Lassila. Inoltre gli sforzi effettuati finora per la sua diffusione riguardano quasi esclusivamente il mondo *desktop*. Risulta quindi semplice capire il perché della mancanza di qualsiasi supporto per dispositivi mobili.

I reasoner menzionati nel paragrafo precedente sono però scritti in Java e sono quindi potenzialmente compatibili alla piattaforma Android.

Purtroppo questo non è comunque sufficiente a poter utilizzare sulla piattaforma strumenti di *reasoning* poiché utilizzano librerie non supportate e talvolta non compatibili. Ad esempio il framework *Jena* dipende da API di Java e da librerie *Apache Xerces* (per la gestione di datatype XML) non incluse nel *classpath* di Android.

Il riadattamento (o *porting*) del software è un’operazione dispendiosa e non era previsto per lo svolgimento del nostro progetto di tesi. Ad ogni modo si è tentata lo stesso questa strada anche perché stimolati dal successo di alcuni sviluppatori. Come per uno sviluppatore Indiano che affermava di essere riuscito

ad effettuare il porting senza problemi (ma che non ha mai risposto alle nostre richieste di informazioni) o di uno sviluppatore Italiano, Fabrizio Giudici, che ha riadattato alcuni moduli del framework Sesame.

Alla fine la soluzione al problema è arrivata da un Ingegnere Informatico, Lorenzo Carrara, del Politecnico di Milano che ha sviluppato AndroJena.

Il framework AndroJena consiste nella modifica del codice sorgente di Jena allo scopo di:

- aggirare le dipendenze dalle API Java
- eliminare le dipendenze da Apache Xerces in modo da evitare il porting di una libreria "pesante" e ridondante (Android ha già una libreria simile integrata)

La modifica ha interessato anche gli strumenti utili ad memorizzare ed interrogare basi di dati semantiche. Grazie alla completezza di questo lavoro abbiamo avuto la possibilità di portare avanti il nostro progetto.

4. Progettazione

4.1 Requisiti Progettuali

L'obiettivo della tesi è la realizzazione, su piattaforma Android, di un framework per il controllo dell'accesso basato su *Proteus*: un framework di politiche di sicurezza fondato su un modello *Context-Aware* Semantico (Toninelli, Montanari, Kagal, & Lassila, 2007). In particolare si vuole progettare un sistema di controllo sensibile al contesto che permetta anche agli utenti meno smaliziati di creare e gestire con semplicità le politiche di sicurezza (*policy*) che ne regolano il funzionamento.

Il modello si basa sul concetto che la comprensibilità delle politiche di sicurezza, in un sistema di controllo dell'accesso, è condizione necessaria per l'usabilità del sistema stesso. Creare un'interfaccia utente di facile utilizzo non è l'unico passo richiesto, occorre altresì creare un livello di astrazione capace di colmare il divario tra le regole pensate dagli utenti (quindi rappresentate secondo il loro processo mentale) e la realizzazione pratica delle stesse. Per questo motivo l'approccio scelto adotta una rappresentazione delle politiche basata sulla semantica, per aiutare gli utenti a comprendere al meglio le politiche e rappresentare al meglio le loro necessità (Toninelli, Montanari, Lassila, & D., 2009).

Dunque il sistema deve poter fornire il supporto per semplificare la gestione della sicurezza in modo che chiunque ne possa comprendere il funzionamento. Inoltre deve essere ben integrato con il resto delle applicazioni della piattaforma Android, in altre parole deve essere possibile accedere alle funzionalità del framework da più punti della piattaforma. Ad esempio nel momento in cui scorriamo la rubrica telefonica per selezionare un contatto ci potrebbe venire in

mente di definirne una politica relativa: all'interno della rubrica stessa dovrebbe essere possibile effettuare questa operazione senza dovere aprire un programma esterno o modificare manualmente file di configurazione.

In generale, i sistemi che richiedono una specifica definizione di politiche, magari anche in un linguaggio particolare, risultano nella maggioranza dei casi o incomprensibili a chi non è avvezzo a materie informatiche, o scomodi e dunque inutilizzati anche per un'utenza specializzata.

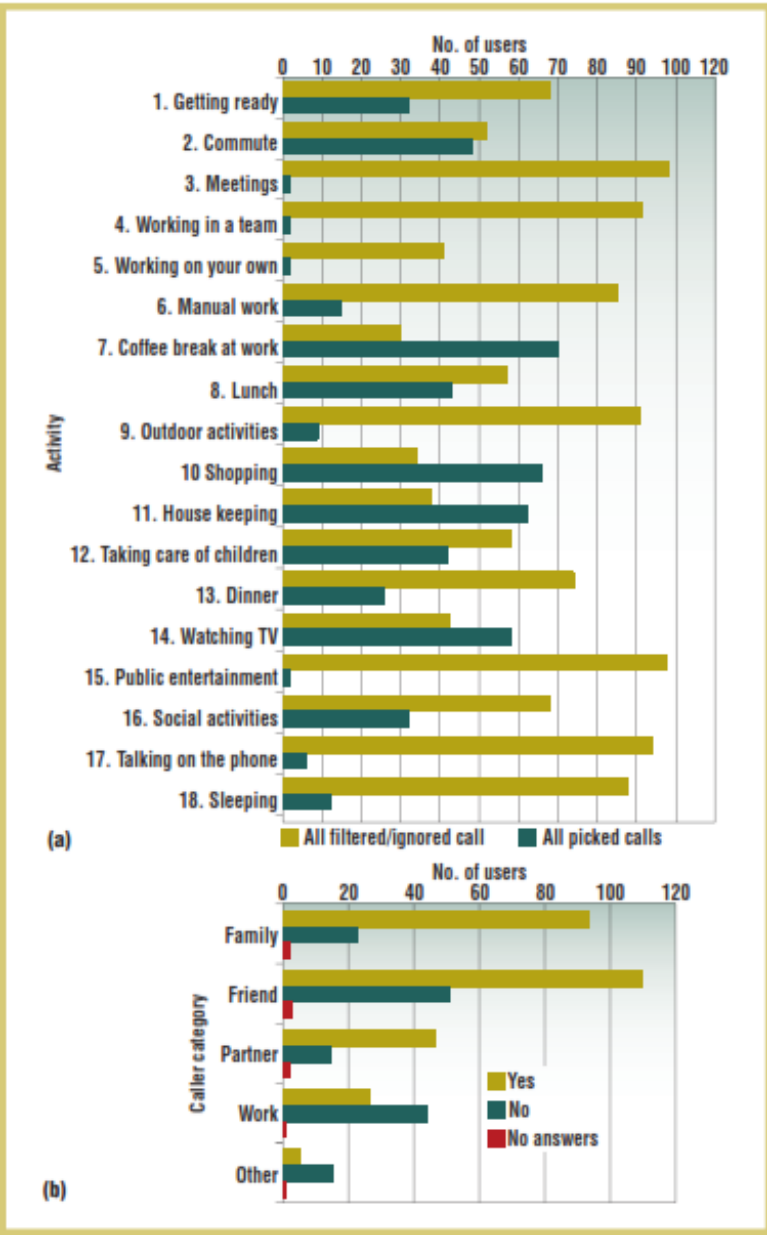
Secondo ricerche effettuate su un campione di utenti è stato rilevato che sistemi di controllo dell'accesso sarebbero particolarmente utili per svolgere alcune operazioni:

- **Condivisione Documenti:** Si potrebbe voler condividere file con tutti i contatti o solamente con una parte di essi.
- **Notifica a terzi delle Attività Personali:** Gestione delle informazioni di stato. Alcune informazioni sensibili, (posizione geografica, attività o stato corrente, ...) potrebbero essere condivise. Trattandosi di dati personali occorre fare scelte oculate.
- **Controllo delle Interruzioni:** Gestione delle interruzioni da parte di terzi. A causa della mancanza di appropriati meccanismi di controllo la maggior parte degli utenti sceglie semplicemente di spegnere il proprio telefono. Quando in realtà sarebbero disponibili ad essere disturbati solo da una certa tipologia di contatti.

Nel corso della tesi ci è occupati prevalentemente del terzo scenario. Studi sul settore hanno appurato che solo poche persone (circa il 30%) gradirebbero un sistema per la gestione automatizzata delle chiamate, mentre il restante 70% avrebbe paura che errori di filtraggio causino più inconvenienti che vantaggi e decidono quindi di gestire da loro le chiamate in arrivo. Risulta altresì evidente come la scelta di come filtrare le chiamate sia fortemente dipendente dall'attività

svolta al momento della decisione, e dalla relazione sociale con la persona che sta effettuando la chiamata.

Figura 9 Grafico delle Ricerche effettuate



Classificando le conoscenze e le attività dell'utente può risultare relativamente semplice definire delle politiche efficienti, ma a volte la volontà

dell'utente può essere difficile da “tradurre” oppure è richiesto un intervento diretto.

Inoltre un utente può trovare difficoltoso definire una strategia che ha in mente nonostante sia tecnicamente possibile, questo a causa della scarsa presenza di strumenti utili per la gestione della sicurezza.

Vi sono però tutta una serie di considerazioni di massima di cui bisogna tenere conto nel corso dello sviluppo del framework per arrivare ad un modello di facile uso per il controllo dell'accesso:

- Le relazioni sociali e le attività svolte dall'utente in quel determinato momento sono fattori cruciali. Il policy-model dovrà essere social-aware per poter valutare i rapporti tra il chiamante e il proprietario dello smart phone al momento dell'*enforcing* (effettiva attuazione della policy).
- Solitamente un utente tende a definire politiche più per gruppi di persone accomunati da caratteristiche sociali che per singoli individui: componenti di una squadra, colleghi di lavoro, amici, parenti, etc.
- Le politiche dovrebbero essere specificabili in relazione all'attività svolta nel momento della definizione. In altre parole durante l'uso di un'applicazione (es. l'agenda degli appuntamenti) relativa ad un particolare contesto (es. evento in agenda) si dovrebbe passare facilmente alla definizione di una politica riferita al contesto (es una politica avente come contesto l'evento in agenda).
- Un utente vede le politiche come una serie di condizioni in cui un evento si può verificare. Queste condizioni dovranno essere espresse sia in un linguaggio “interno” ad uso esclusivo dell'applicazione, sia in un formato “testuale”, human-readable per facilitare la comprensione ad un utente inesperto. Inoltre deve essere possibile recuperare e raggruppare le politiche da diversi punti di vista.

4.1.1 Socially Aware Policy Model

Per definire una politica l'approccio migliore è porsi alcune domande per capire cosa si intende regolamentare:

- Che risorsa deve essere acceduta e quali sono le sue caratteristiche?
- Chi richiede l'accesso alla risorsa e quali sono le sue caratteristiche?
- In quale contesto viene richiesto l'accesso alla risorsa?
- Come viene acceduta la risorsa?

Come si può facilmente intuire, a livello mentale, le dimensioni who/what/where/how sono tra loro strettamente correlate e solitamente fanno tutte parte di una politica. L'obiettivo è fare in modo che il modello rispecchi totalmente il modo di pensare dell'utente e supporti questa modalità di definizione di politiche *cross-dimensional*. Ovvero non bisogna definire a priori tutti i componenti della policy e assegnare loro degli attributi, delle caratteristiche o cose di questo genere. Una policy viene creata sul momento come una relazione di differenti elementi interconnessi.

Quando è il momento di effettuare il vero e proprio *enforcing*, è necessario reperire delle informazioni aggiuntive (sull'identità del chiamante) per poter controllare se la situazione del contesto attuale rispecchia una politica definita e salvata nel sistema.

Per capirci: se si crea una politica che recita “*When I am in a **meeting**, my phone can **ring** if a **client** calls me*” (Quando sono in Riunione, il mio telefono può squillare se mi chiama un cliente), alla ricezione di una chiamata non basterà controllare l'identità del chiamante, ma sarà necessario valutare se si tratta di un amico, di un parente o di un cliente per poter applicare la politica corretta. Inoltre lo stato dovrà essere impostato in “Meeting”.

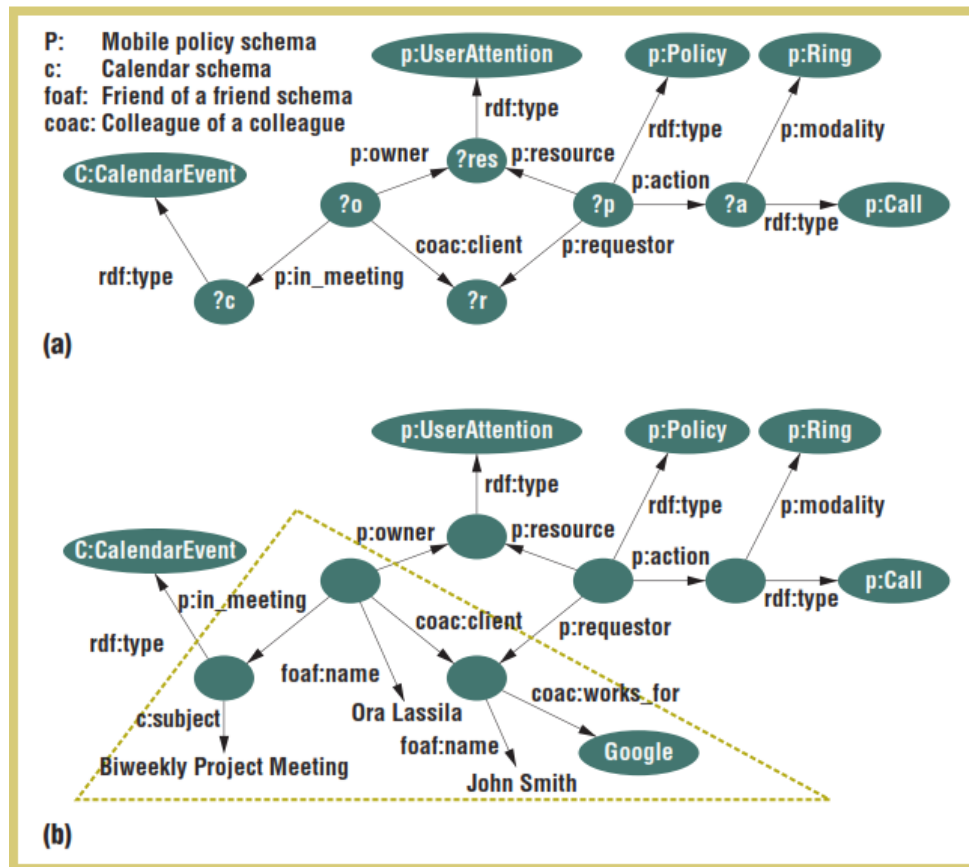
Per rappresentare al meglio questo tipo di politiche si è preso in considerazione l'utilizzo di strutture dati di tipo semantico, come lo standard RDF.

Resource Description Framework è un standard definito per il semantic-web che permette di rappresentare informazioni come grafi orientati ed etichettati. Il modello proposto è generico e può essere utilizzato per descrivere fenomeni del mondo reale. Le strutture dati sono flessibili, estendibili e facilmente modificabili, rispecchiando in tal modo i processi che si sviluppano all'interno della mente umana. Inoltre esso permette associazioni tra elementi formali della semantica e dati veri e propri rendendo possibile processi di *reasoning* su concetti e relazioni.

Ad alto livello consideriamo la struttura di una politica come un insieme di attributi con valori predeterminati. Compito di questi attributi è definire e descrivere il what/who/how e la dimensione contestuale (when/where/why). Ovviamente questi valori possono essere predefiniti o varibili con valore illimitato o limitato all'interno di un range. Affinchè una politica sia applicabile è necessario che i valori che definiscono lo stato attuale del sistema, facciano match con quelli definiti e fissati all'interno della politica stessa.

Presentiamo uno schema RDF di una possibile politica. Essa è strutturata come un grafo orientato ed etichettato: la label sulle frecce indica la relazione che lega due nodi, mentre quella sul nodo indica l'istanza o la classe di cui si tratta. La **Figura 10(a)** rappresenta la politica: “*When I am in a **meeting**, my phone can **ring** if a **client** calls me*”.

Figura 10 Esempio di Grafo delle Politiche



E' possibile riconoscere all'interno del grafo tutte le componenti di cui parlavamo precedentemente: **p:requestor** è la parte di who, **p:resource** è il what, **p:action** l'how. Inoltre definiamo informazioni di contesto inserendo delle relazioni che specificano alcune caratteristiche di una risorsa (ad esempio **p:in_meeting** lega il proprietario del telefono ad un evento del calendario per definire la parte della politica "When I am in a Meeting").

Una politica viene rappresentata come un set di statement RDF o di triple SPARQL (una tripla è un elemento composto da tre elementi: soggetto-predicato-oggetto), in cui ogni statement rappresenta un attributo e il suo relativo valore. SPARQL è un linguaggio di query, molto simile ad SQL, che serve per interrogare basi dati di tipo RDF. Si tratta di un linguaggio identico ad RDF con la

sola eccezione che ciascun elemento della tripla può essere una variabile. Tramite questo linguaggio è possibile effettuare diversi tipi di operazioni (filter, join, condition, etc..) sulla base dati di tipo RDF.

Facendo riferimento alla **Figura 10(a)**, possiamo esprimere lo statement “*the Resource protected by the policy is the User’s Attention*” (La Risorsa protetta dalla politica è l’Attenzione dell’Utente) utilizzando le seguenti triple SPARQL:

(?p, rdf:type, p:policy)

(?p, p:resource, ?res)

(?res, rdf:type, p:UserAttention)

Queste triple formano una sorta di *graph template*, dove alcuni nodi rappresentano variabili non ancora istanziate. Possiamo leggere la policy SPARQL rappresentata dalla **Figura 10(a)** come:

if [a policy (?p) for resource (?res) has requestor (?r)]

and [the resource (?res) has owner (?o) who is in a meeting (?c)

and has client (?r)]

then [action (?a) of policy (?p) is permitted; thus

for a call (?a) from the requestor (?r) the phone will ring]

La **Figura 10(b)** (vedi triangolo tratteggiato) illustra invece un esempio dello stato corrente del sistema realizzato tramite un set di statement che fa match

con la policy della **Figura 10(a)**. Sono presenti dei *blank nodes*, ovvero nodi non associati ad alcun particolare Uri (leggi Risorsa), che realizzano la necessaria connettività tra le varie parti del sistema.

4.1.2 Model Usability

Come possiamo comprendere dall'esempio specificato precedentemente, il modello proposto risulta essere intuitivo e, cosa importante, supporta la modellazione di politiche *Social-Aware*. L'utilizzo di una modellazione a grafo permette inoltre una navigazione multidirezionale (partendo da requestor, resource, etc.). Il framework dovrà inoltre essere fornito di un tool per permettere la navigazione del grafo e la creazione delle policy: un'interfaccia utente (GUI) darà la possibilità di inserire e selezionare gli elementi in maniera semplificata per poi essere tradotti in linguaggio SPARQL internamente senza che l'utilizzatore debba avere a che fare con dettagli di basso livello. Le politiche dovranno poter essere modificate o eliminate, il tutto nel modo più semplice ed immediato possibile.

Tra i vantaggi di utilizzare un linguaggio oramai standard come SPARQL è che le politiche non sono vincolate ad un particolare sistema operativo, linguaggio di programmazione o struttura hardware. Esse possono essere trasferite, copiate, distribuite e, avendo a disposizione un sistema compatibile, possono essere tranquillamente utilizzate per gestire l'accesso su un altro dispositivo. In altre parole per utilizzare le stesse politiche su un dispositivo differente è sufficiente importare le stesse senza dover cambiare nulla tra le informazioni di sistema. Questo perché le regole fanno riferimento ad entità astratte, ed è compito del sistema interpretarle.

4.2 Analisi del Sistema

Nel progetto ci si è soffermati sulla gestione della risorsa “**User Attention**”. Quindi d’ora in poi si farà riferimento al comportamento del framework in occasione di una chiamata in arrivo ed in presenza di un determinato contesto.

Data la descrizione del sistema richiesto possiamo riassumere i requisiti in questo modo:

- Gestione delle politiche (creazione/modifica/cancellazione)
- Gestione del contesto (sociale/status)
- Definizione politiche tramite integrazione in altre applicazioni
- Enforcing delle politiche

Vi sono inoltre altri requisiti non funzionali ma non trascurabili, come ad esempio la struttura dell’interfaccia grafica (GUI) che dovrà risultare estremamente semplificata e di facile utilizzo. Anche l’efficienza sarà un fattore rilevante: le modalità di reperimento delle informazioni sociali e di contesto, le elaborazioni interne del sistema non dovranno comportare consumi eccessi per il dispositivo. Bisogna sempre tenere presente che si sta producendo software per smartphone che, pur essendo abbastanza performante, ancora non è paragonabile ad un computer vero e proprio. Limiti di spazio, di memoria occupata, di connettività, di consumo energetico diventano fattori principali nella fase di analisi e poi in quella di progettazione del prodotto.

4.2.1 Casi d’Uso

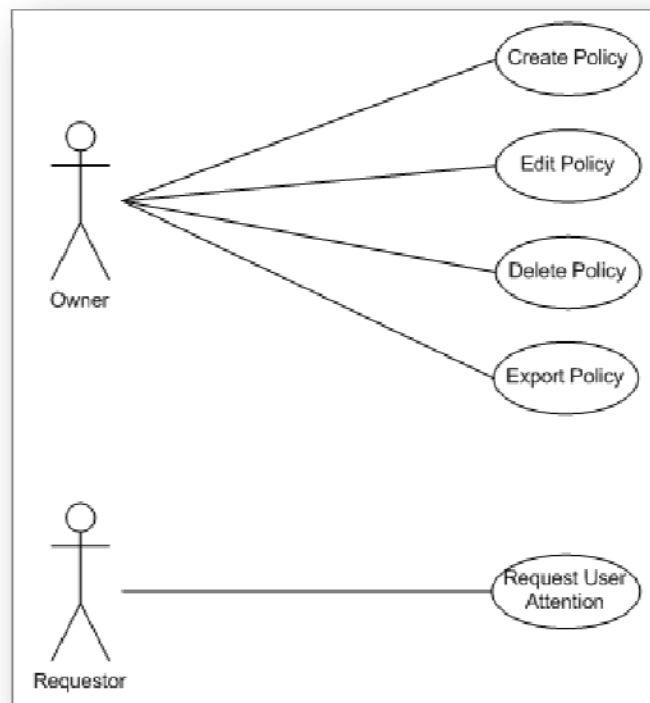
Una volta definito cosa vogliamo realizzare è necessario comprendere come il sistema interagirà con il mondo esterno. Ovvero quali sono i casi d’uso che identificano l’interazione del sistema con l’utente o altri attori esterni.

Abbiamo considerato solamente due attori esterni al sistema: l'utente (**Owner**, proprietario, della risorsa) e il richiedente della risorsa (**Requestor**).

Il primo è quello che andrà a definire le policy. Potrà anche vedere quelle attualmente presenti sul dispositivo, effettuare una ricerca in base a diversi criteri e potrà attuare operazioni di modifica, cancellazione o esportazione.

Il secondo attore, *requestor*, non è altro che la chiamata in arrivo che richiede al sistema l'accesso alla risorsa *User Attention*. Il framework dovrà recuperare le informazioni necessarie (la relazione sociale della persona chiamante) per poter processare la richiesta correttamente.

Figura 11 Casi d'Uso



Le informazioni necessarie per effettuare l'*enforcing*, a fronte di una richiesta, non sono dichiaratamente localizzabili sul dispositivo: esse potrebbero trovarsi su un server di un social network, o su un database remoto. Sono ovviamente tutte scelte progettuali-implementative.

4.2.2 Scenari Operativi

Gli scenari operativi principali risultano essere due:

- L'utente interagisce con il framework.
- Arriva una richiesta di accesso ad una risorsa.

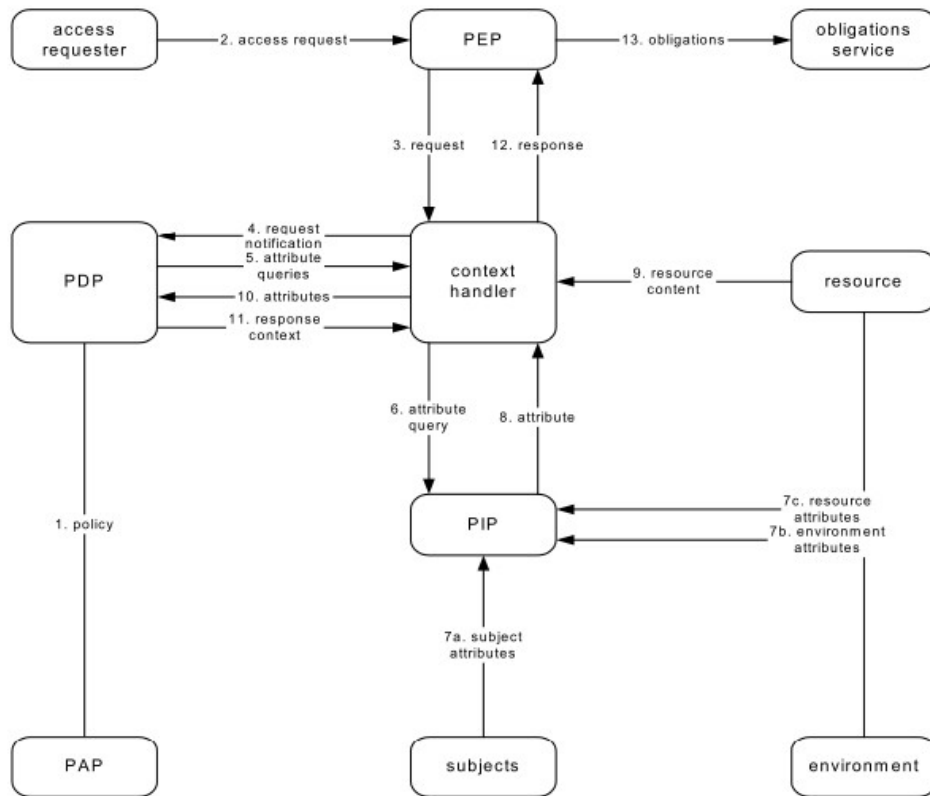
Un esempio del primo caso può essere l'utente che utilizza un'applicazione per effettuare alcune operazioni, come salvare un appuntamento o aggiornare le informazioni di un contatto: direttamente dall'applicazione stessa potrebbe voler fare delle operazioni sulle policy. Una volta completate le modifiche, le policy vengono salvate sull'archivio del sistema e il controllo ritorna all'applicazione iniziale. L'utente può procedere con le proprie attività.

Il secondo scenario ipotizzato prevede la valutazione della richiesta da parte del sistema. Nel caso in esame si tratta di una chiamata che tenta di accedere alla risorsa User Attention. All'arrivo della chiamata, la parte di sistema incaricata dell'*enforcing* avrà il compito di intercettarla ed il resto del sistema si occuperà di prendere una decisione in base allo stato attuale ed alle informazioni relative alla chiamata stessa.

4.2.3 Struttura Logica

Chiudiamo questa sezione di analisi presentando una tipica struttura logica applicata ad un sistema di controllo degli accessi di tipo Context-Aware.

Figura 12 Esempio di Schema logico di un sistema Context-Aware



Nella figura si individuano i componenti principali che interagiscono nel sistema:

Risorse: Elementi amministrati dal sistema. Il prototipo prevede solamente la User Attention, ma nulla ci vieta di estendere le funzionalità alla gestione di files o altri tipi di risorse.

Politiche: Direttive assegnate al sistema. Esso infatti si comporterà in base alle policy che avremo definito. Queste sono conservate all'interno un repository (PAP) e vengono utilizzate per le operazioni di gestione o per l'enforcing.

Policy Administration Point (PAP): E' la parte logica del sistema che ha il compito di gestire le policy. Deve permettere azioni di aggiunta, modifica, cancellazione, browsing e ricerca. Altra sua responsabilità è la modalità di salvataggio, la gestione della consistenza e della ridondanza.

Policy Decision Point (PDP): E' la parte logica del sistema che, una volta recuperate le informazioni necessarie e le policy dal PAP, effettua la vera e propria decisione. Nel nostro caso viene valutata la presenza o meno di una policy che collida con lo stato attuale del sistema.

Policy Enforcement Point (PEP): E' la parte logica del sistema che ha il compito di intercettare le richieste dei partecipanti e di richiedere al PDP una valutazione per un dato caso. Sua è anche la responsabilità di effettuare l'enforcing quando una decisione viene ufficialmente presa dal sistema.

Policy Information Point (PIP): E' la parte logica del sistema che ha il compito di recuperare le informazioni esterne necessarie alla definizione del contesto.

Context Handler: E' la parte logica del sistema che si interpone tra il PEP ed il PDP. Si occupa della gestione di tutte le informazioni di contesto e delle risorse.

Al solo scopo di introdurre i concetti alla base del funzionamento di un sistema di controllo degli accessi Context-Aware, analizziamo gli eventi rappresentati in figura. Notiamo come la richiesta del partecipante venga intercettata dal PEP, il quale a sua volta richiede ad un altro componente (il Context-Handler) se deve permettere o meno l'esecuzione dell'operazione richiesta. Il Context Handler, una volta recuperate le informazioni riguardanti la Risorsa, il Contesto e il Richiedente, passa questi dati al PDP che avrà il compito di effettuare la valutazione. Il risultato prodotto dal PDP sarà passato al PEP che in tal modo saprà se deve permettere o bloccare l'azione richiesta.

4.3 Implementazione del Sistema

Come menzionato precedentemente nel capitolo “Stato dell’Arte”, la piattaforma utilizzata per sviluppare il nostro framework (o meglio *middleware*) è Android. Risulta utile fare una breve digressione su alcune sue caratteristiche, in modo da comprendere i motivi che hanno guidato le scelte durante l’implementazione.

Il livello applicativo di Android è costituito da svariate Applicazioni in grado di interagire tra loro. Ogni applicazione costituisce un componente autonomo che può essere aggiunto ed integrato al sistema operativo con estrema flessibilità. Questa potenziale espandibilità dell’ambiente applicativo non riguarda però la singola applicazione, il quale rimane in ogni caso un componente autonomo. Dunque, al contrario della piattaforma in sé, un’applicazione non può essere estesa quando è ormai installata nel sistema, per raggiungere tale obiettivo occorre modificarne il codice sorgente e installarla nuovamente.

Tenendo queste peculiarità in mente passiamo a considerare i requisiti progettuali riguardo all’integrazione del *middleware* all’interno delle applicazioni di Android. Facendo riferimento proprio all’esempio della Rubrica Contatti proposto all’inizio di questo capitolo (la possibilità di definire una politica relativa ad un contatto direttamente dall’applicazione Rubrica), l’implementazione del sistema ha anche coinvolto la modifica del codice sorgente dell’applicazione, estendendone le funzionalità per i nostri scopi.

Partendo dall’analisi delle responsabilità abbiamo suddiviso il sistema in componenti che hanno il compito di fornire e realizzare le varie funzionalità. Quindi il sistema si compone di cinque componenti e la libreria necessaria per utilizzare il formato RDF. Nelle sezioni successive li analizzeremo in maniera più specifica:

GUI: Interfacce grafiche necessarie all'utente per gestire le politiche del sistema.

Policy Provider: E' il componente che si occupa di contenere le policy e fare in modo che siano disponibili a chi le richiede.

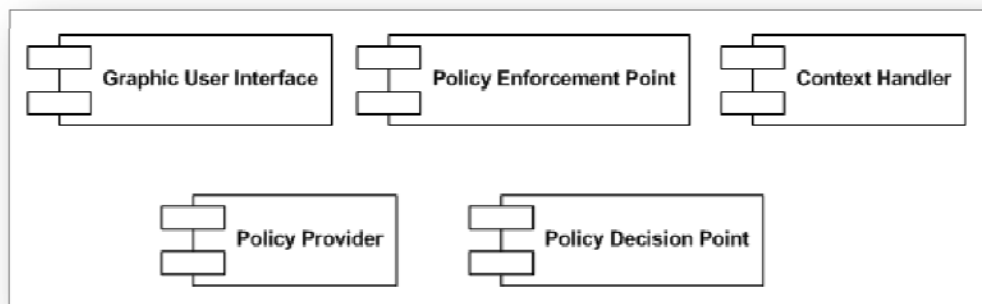
Context Handler: E' il componente che ha il compito di recuperare le informazioni di contesto e sociali. Esso interagirà con dei server esterni o database per fornire tutte le informazioni necessarie per le valutazioni.

PEP (Policy Enforcement Point): Si tratta del componente che si occupa di intercettare le richieste che dobbiamo processare (In questo caso le chiamate in ingresso). Al termine della valutazione del sistema dovrà effettuare il vero e proprio enforcing.

PDP (Policy Decision Point): Si tratta del componente incaricato di valutare la richiesta di accesso alla risorsa User Attention, e decidere quale azione fare eseguire al PEP.

AndroJena: si tratta della libreria **Jena** (*Semantic Reasoner*) modificata per poter essere utilizzata sulla piattaforma Android.

Figura 13 Componenti del Sistema



4.3.1 Graphic User Interface

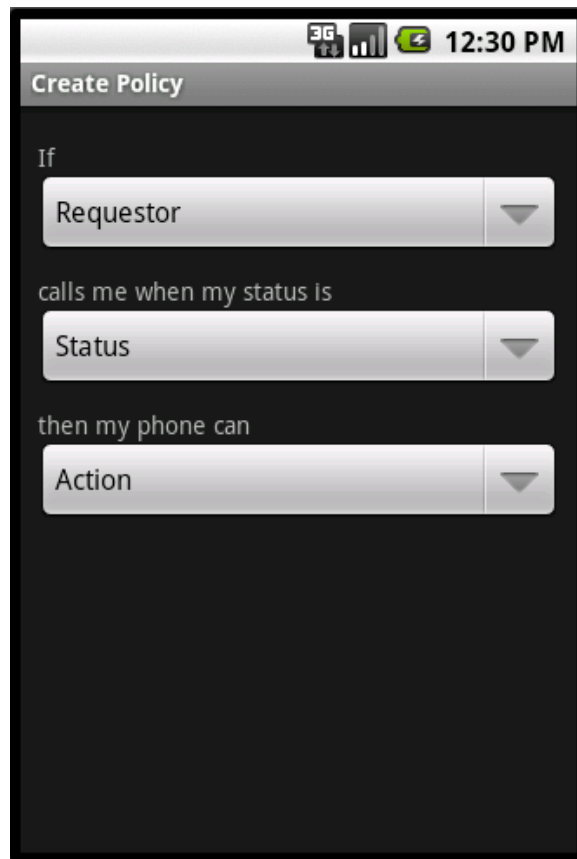
Questo componente è composto da interfacce grafiche realizzate basandosi sul presupposto che l'applicazione non dovrà essere preclusa all'utente inesperto.

Le funzionalità offerte permettono di:

- Creare delle politiche.
- Modificare/Cancellare politiche.
- Effettuare ricerche sulle politiche.

Di seguito un esempio di definizione di una politica. Il nodo "Requestor" può riferire sia ad un nodo preesistente, ovvero di default del sistema (come *client*, *colleague*, *friend* ecc.), oppure direttamente all'identificativo di un contatto in rubrica.

Figura 14 Interfaccia per la creazione di una Policy



The screenshot shows a mobile application interface titled "Create Policy". At the top, there is a status bar with "3G", signal strength, battery level, and the time "12:30 PM". Below the title bar, the interface is divided into three sections, each with a label and a dropdown menu:

- If**: The dropdown menu is labeled "Requestor".
- calls me when my status is**: The dropdown menu is labeled "Status".
- then my phone can**: The dropdown menu is labeled "Action".

4.3.2 Il *Semantic Reasoner* AndroJena

Come abbiamo già accennato precedentemente, la libreria AndroJena non è altro che la versione per Android della libreria Jena.

Questo strumento per *Semantic Reasoning* permette di:

- Creare un grafo (ovvero modello) **RDF** a partire da file esterni.
- Creare un grafo RDF tramite le proprie API Java.
- Manipolare un grafo RDF: modificando nodi singoli o triple intere.
- Interrogare tramite query **SPARQL** il grafo RDF: la query può essere effettuata per richiedere un sottoinsieme dei grafi che corrispondono alle caratteristiche specificate dalla query stessa, oppure per verificare l'esistenza del grafo specificato dalla query.
- Memorizzare un grafo o un insieme di grafi in modo permanente (su disco) grazie al componente **TDB**. Il Triple DataBase di Jena è un componente per memorizzare in modo persistente i grafi. Si tratta di un sotto-sistema di memorizzazione ad alte prestazioni, ed interamente basato su Java. Un archivio TDB è creato e gestito tramite le API di Jena. Viene solitamente utilizzato quando si richiede velocità di elaborazione e la memorizzazione su una singola macchina (quindi non è richiesto un database).

4.3.3 Policy Provider

Il Policy Provider ha la responsabilità della consistenza delle policy. Nel momento in cui aggiungiamo, modifichiamo, creiamo una nuova policy, le informazioni nel repository devono essere aggiornate. Quando poi il dispositivo viene spento deve fare in modo che le informazioni siano salvate e disponibili al momento del riavvio dello smartphone.

Le sue mansioni principali sono:

- Memorizzare la Policy ricevuta dalla GUI in formato RDF.
- Memorizzare la Policy in formato SPARQL dopo averla convertita dal formato RDF.
- Effettuare ricerche sul DataBase delle Policy.

Per **memorizzare** la policy creata il Policy Provider trasforma le informazioni ricevute dalla GUI (Requestor/Status/Action) in triple RDF così come descritte precedentemente nel paragrafo 4.1.1.

L'archiviazione delle policy in formato RDF rende più facili le operazioni di **ricerca** di una determinata politica. Infatti il sistema di interrogazione del grafo RDF permette di effettuare una query incentrata su qualsiasi nodo del grafo. Quindi ad esempio, cercando il nodo "cliente" (`coac:client`) o il nodo "squilla" (`p:Ring`) otterrò comunque la politica *Quando sono in Riunione, il mio telefono può squillare se mi chiama un cliente*.

La **conversione** nel formato SPARQL risulta invece utile nella effettiva valutazione delle richieste di accesso. Ne tralasciamo il motivo per il momento, per una spiegazione esaustiva si faccia riferimento alla sezione relativa al Policy Decision Point.

Riguardo alla memorizzazione persistente delle politiche in versione RDF è stato utilizzato il componente di Jena **TDB** (vedi par. 4.3.2). Questo ci permette di effettuare query sull'archivio delle politiche in modo molto più efficiente. Supponendo infatti la memorizzazione delle politiche su file avremmo avuto ritardi nel caricamento nell'ambiente Jena delle politiche. Mentre con un'ipotetica memorizzazione su DataBase classico non sarebbe stato possibile effettuare query.

Esempio di creazione di un dataset TDB

```
Model model = TDBFactory.createModel(directory)
Resource johnSmith
    = model.createResource(personURI)
        .addProperty(VCARD.FN, fullName)
        .addProperty(VCARD.N,
            model.createResource()
                .addProperty(VCARD.Given, givenName)
                .addProperty(VCARD.Family,
                    familyName));
```

Le query SPARQL vengono memorizzate sotto forma di stringa in un DataBase SQLite accessibile al solo Policy Provider. Questo DataBase risulta nativo in Android. Si tratta di un embedded relational database management system che differentemente da sistemi client-server, non è implementato come processo stand-alone con cui l'applicazione comunica. Al contrario esso è inserito all'interno dell'applicazione stessa. La libreria SQLite viene caricata dinamicamente, e l'accesso ai dati è effettuato tramite l'invocazione di metodi. Tutto ciò permette di ridurre tempistiche di accesso al database rispetto a modelli multi-processo. Il database viene salvato sottoforma di singolo crossplatform file sulla macchina che utilizza il programma.

Il Policy Provider viene utilizzato dagli altri componente tramite un *lightweight mechanism* per eseguire RPC supportato da Android: AIDL. Una RPC permette, tramite una chiamata locale, l'esecuzione di un metodo remoto a carico di un altro processo (per ulteriori delucidazioni sull'argomento si confronti il capitolo relativo ad Android). Ecco alcuni metodi che il componente espone:

```
boolean        createPolicy(String[] policy)
String[]       searchPolicy(String[] policy)
```

```
String[]      browsePolicy()
```

4.3.4 Context Handler

Il Context Handler ha la responsabilità di tener aggiornate le informazioni relative al contesto.

Lo stato del sistema ed il suo contesto viene rappresentato tramite ontologie RDF. Le ontologie sono memorizzate nel sotto-sistema TDB del framework Jena. In questo modo si garantisce velocità e flessibilità nella manipolazione dei dati (grafi).

Contestualmente al recupero delle informazioni di contesto il Context Handler si occupa di aggiornare il grafo RDF del sistema. Purtroppo le fonti da cui recuperare queste informazioni non forniscono dati standard, quindi è stata prevista un'interazione dell'utente per associare manualmente un significato semantico ai dati ottenuti.

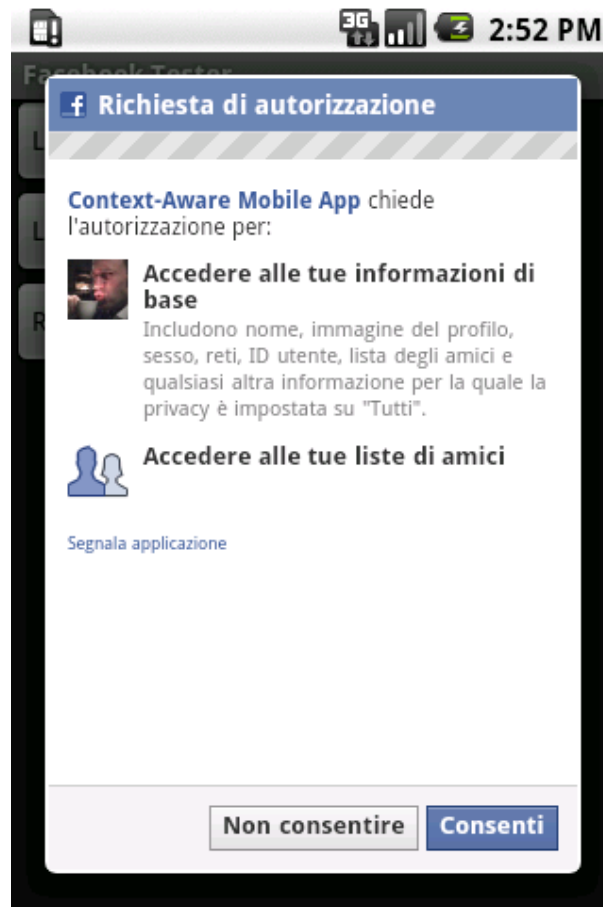
Riassumiamone quindi le funzioni principali:

- Recupero informazioni sociali da Facebook e LinkedIn (Gruppi e Relazioni Sociali)
- Ricezioni eventi del Sistema Operativo per l'aggiornamento del contesto.

Un aspetto molto importante di questo componente è la capacità di recuperare informazioni anche dai Social Network come **Facebook** e **LinkedIn**.

Grazie alle API (ufficiali e non) fornite da queste piattaforme possiamo interrogare direttamente i loro servizi per conoscere i gruppi ed altre informazioni relative all'utente.

Figura 15 Accesso del Sistema al Social Network Facebook



La parte più ardua sarà far corrispondere tali informazioni con i contatti presenti nel telefono (particolare importante visto che il numero telefonico è l'unico modo per riconoscere la persona coinvolta). Parte dei contatti telefonici sarà associata ai profili dei Social Network in modo automatico se i nomi corrispondono. In caso contrario sarà richiesto l'intervento dell'utente tramite una Interfaccia, istanziabile da quella principale, dedicata ad associare questi dati. Inoltre vi è un'altra interfaccia simile dedicata ad associare i gruppi presenti sui Social Network e le classificazioni RDF (come *client*, *colleague*, *friend* etc.). Il Context Handler si occuperà di mantenere queste associazioni in un DataBase interno.

Supponendo quindi di aver completato l'associazione dei contatti della Rubrica del telefono con quella del Social Network, riportiamo ora l'esempio in

cui si ottenga che la persona “John Smith” sia un cliente. Il grafo relativo al sistema verrà così aggiornato:

```
(p:User, coac:client, :clientnode)
(_:clientnode, foaf:name, John Smith)
(_:clientnode, p:ID, John Smith)
```

Dove ID è l’identificativo univoco assegnato dal dispositivo ad un contatto della Rubrica. Questa informazione è essenziale per la valutazione della chiamata in ingresso.

Le **attività** dell’utente, relative all’agenda appuntamenti, sono aggiornate direttamente sfruttando una funzionalità di Android: *Broadcast Receiver*. Tutti componenti Android comunicano tra di loro tramite messaggi chiamati *Intent*. Ogni evento all’interno del telefono ne scatena uno, dall’arrivo di un sms alla notifica di batteria in esaurimento. Sfruttando quindi questa peculiarità possiamo captare ogni appuntamento in agenda, recuperarne le informazioni relative dal database Android ed infine aggiornare il grafo RDF relativo allo stato attuale.

Vale la pena notare che questa implementazione, rispetto al modello classico in cui le informazioni di contesto vengono recuperate al momento della valutazione dell’accesso, permette di implementare un sistema che è “cosciente” del contesto in cui opera (*Context-Aware*) in maniera continuativa. Inoltre mantenere lo stato attuale sempre aggiornato, porta ad accelerare la fase di “Decision” e, di conseguenza, di “Enforce”. Questo perché si riducono le informazioni da inserire per aggiornare lo stato attuale, e il tempo di reazione a fronte dell’evento “chiamata in arrivo” è dato dal solo tempo di valutazione della policy.

Il Context Handler riceve infine una chiamata dal PEP (al momento di una richiesta d'accesso) contenente l'identificativo del chiamante (**requestor**), a questo punto non rimane che recuperarne le informazioni relative dal suo database ed aggiornare il grafo del sistema.

```
boolean          updateRequestor(String id)
```

```
(p:Requestor, foaf:name, name)
```

4.3.5 Policy Enforcer Point

Si tratta di tutte le classi che si occupano di gestire l'enforcing dopo aver ricevuto la richiesta d'accesso, ovvero la Chiamata in Entrata.

Il componente implementato si occupa della gestione delle chiamate in arrivo, ma estenderlo per la gestione degli sms o altro è un problema triviale.

Per la realizzazione di questo componente sono state utilizzate delle API di Android non ufficiali. Infatti la natura open source del sistema operativo ha reso possibile l'uso di strumenti che altrimenti sarebbe stato impossibile utilizzare.

Per accedere alle API non ufficiali si è utilizzato le proprietà di *reflection* di Java per poter ottenere accesso al servizio `ITelephony` ed avere la possibilità di poter terminare una chiamata.

```
Class c = Class.forName(tm.getClass().getName());  
Method m = c.getDeclaredMethod("getITelephony");  
m.setAccessible(true);  
ITelephony telephonyService = (ITelephony) m.invoke(tm);
```

```
telephonyService.endCall();
```

L'Enforcer Point è in grado di:

- Intercettare una chiamata in arrivo ed innescare il meccanismo di valutazione.
- Ricevere istruzioni riguardo all'azione da intraprendere a fronte di una richiesta (suoneria / muto / vibrazione / rifiuta)

Per un'estesa illustrazione del funzionamento a tempo di esecuzione si veda il paragrafo relativo al comportamento del Sistema.

4.3.6 Policy Decision Point

Questo componente si fa carico della valutazione vera e propria delle politiche di sicurezza.

Al verificarsi dell'evento "Chiamata in Arrivo", il PDP deve decidere quale azione far eseguire al Policy Enforcement Point.

Quando riceve una richiesta di accesso da parte di un determinato "Requestor" (le cui informazioni nel sistema sono state prontamente aggiornate dal Context Handler) il PDP recupera le policy a lui relative richiedendole al PAP che restituisce, in caso di riscontro, le policy in formato SPARQL ed il codice dell'**azione associata**.

SPARQL è un linguaggio di query per interrogare dataset – grafi – RDF (vedi par. 4.1.1). Per verificare quindi che nell'ontologia che rappresenta il

sistema vi siano le condizioni necessarie per applicare una policy, è sufficiente eseguire la query corrispondente alla policy.

Il *match* della query indica, in altre parole, che le condizioni espresse nella policy sono verificate perché corrispondenti allo stato attuale.

In caso di match il PDP comunicherà al PEP il codice dell'azione da eseguire, in caso contrario comunicherà il codice corrispondente a “nessuna azione”.

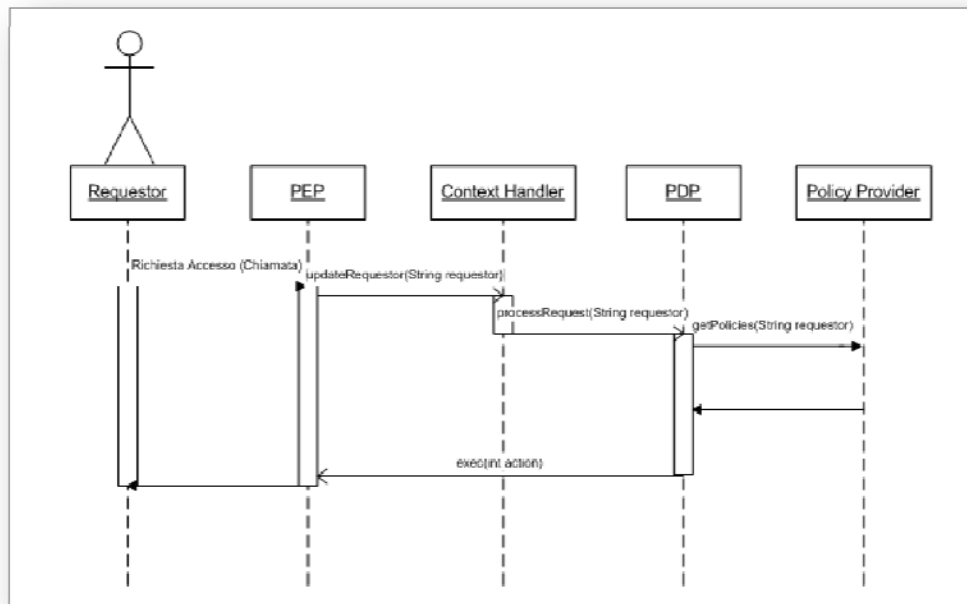
4.3.7 Comportamento

Come già evidenziato nel paragrafo “Casi d’Uso” possiamo individuare lo scenario in cui l’utente interagisce col Sistema per la gestione delle politiche, e quello in cui vi è un attore esterno che richiede l’accesso alla Risorsa UserAttention.

Il primo caso non presenta caratteristiche rilevanti, e quelle generali sono state coperte nei precedenti paragrafi.

Risulta invece interessante presentare il comportamento del sistema a fronte della richiesta di accesso, da parte dell’attore esterno Requestor.

Figura 16 Sequence Diagram



Si parte dalla richiesta del Requestor di accedere alla risorsa, ovvero arriva una chiamata in entrata. Tale richiesta viene intercettata dal PEP grazie al *listener* di eventi di sistema al suo interno: **PhoneStateListener**.

Nel momento in cui intercetta una chiamata il **PEP** tiene in sospeso la richiesta fino all'avvenuta valutazione della stessa da parte del sistema, nel caso in esame questo si traduce nel silenziare immediatamente chiamata in entrata.

Esegue poi un *lookup* veloce, supportato da Android, nel DataBase dei contatti per recuperare l'identificativo univoco del chiamante. (In Android viene associato un ID univoco ad ogni contatto presente nell'archivio, e ciò torna molto utile per la navigazione tra i dati). Quindi inoltra la richiesta di accesso al Context Handler includendo l'identità del richiedente. La chiamata fatta al Context Handler è asincrona (vedi par. 3.1.3), quindi dopo averla effettuata il PEP si mette in attesa di una risposta riguarda all'azione da eseguire.

Il **Context Handler**, ricevuta la richiesta con l'identificativo aggiorna quindi il nodo `p:Requestor` nel grafo RDF, e recupera la relazione sociale associata.

In altre parole scopre se ad esempio nel sistema è associato un nodo del tipo `coac:client` o `coac:colleague`. Fatto questo inoltra la richiesta al PDP con allegato l'ID unitamente alle nuove informazioni ottenute.

Una volta ricevuta la richiesta il **PDP** ha bisogno di accedere alle policy presenti in archivio per effettuare la valutazione. Richiede dunque al PAP l'elenco di tutte le politiche relative al richiedente.

Il **PAP** a questo punto effettua una ricerca nell'archivio delle policy. Abbiamo già discusso precedentemente sulla flessibilità della ricerca su grafi RDF, su cui è possibile fare query rispetto ad un relativo pattern e recuperare tutti i grafi che lo contengono. La ricerca effettuata dal PAP è una ricerca su un solo nodo, il non `p:Requestor` delle policy. Cercherà quindi sia le policy che contengono esplicitamente l'ID del chiamante, che quelle che contengono il contesto sociale associato. Nel caso si abbiano diverse policy definite rispetto all'ID e relazione sociale allora verrà data priorità a quelle definite che specificano ID. Tale priorità viene realizzata implicitamente dall'ordine con cui si restituiscono le policy al PDP. Il PAP restituisce il risultato come lista di query SPARQL con il codice di azione associato.

Il **PDP** riceve dal PAP la lista di policy (query SPARQL) con il codice di azione associato. A questo punto non rimane che eseguire le query, una ad una, per trovare quale policy può essere applicata allo stato attuale. Questa esecuzione viene eseguita nell'ordine con cui si riceve la lista.

Appena viene trovato un riscontro positivo l'esecuzione viene fermata (non ha senso continuare) e viene inviato il codice dell'azione da eseguire al **PEP**.

5. Conclusioni e Sviluppi Futuri

Obiettivo della tesi era la realizzazione, su piattaforma Android, di un framework per il controllo dell'accesso di tipo *Context-Aware* Semantico. In particolare, oltre all'implementazione del sistema stesso, si doveva considerare con attenzione la semplicità di utilizzo del framework in modo da venire incontro anche agli utenti meno esperti.

Le tecnologie offerte da Android sono state fondamentali per il conseguimento degli obiettivi fissati. Allo stesso modo il framework Androjena ci ha permesso di usufruire dell'enorme potenzialità del mondo Semantico, prima relegato al solo ambiente *desktop*. Inoltre la larga diffusione dei Social Network presi in considerazione ci ha permesso di sfruttarne gli strumenti di sviluppo (SDK) con una certa facilità.

Al termine del percorso possiamo dire di aver sviluppato un prodotto che rispecchia fedelmente le caratteristiche desiderate. Siamo riusciti a valutare il contesto sociale dell'utente. Abbiamo integrato, anche se in modo poco ortodosso, le funzionalità del sistema all'interno della piattaforma Android. E' stato presentato all'utente un sistema di facile utilizzo che permette di gestire in maniera semplice il controllo dell'accesso

Un ulteriore passo verso l'integrazione nel sistema potrebbe essere l'estensione del Calendario Eventi. Purtroppo quando si intraprende la strada in cui si sostituisce l'applicazione ci si scontra con un problema. Android è una piattaforma *open source*, ed in quanto tale tutti i produttori di smartphone che la scelgono possono decidere se personalizzarla o meno. Questo porta ad avere sul mercato dei dispositivi con lo stesso sistema operativo, ma con diverse interfacce

utente e programmi applicativi. Infatti il tipo di applicazione che abbiamo esteso per il nostro progetto, potrebbe essere presente su alcuni dispositivi nella versione personalizzata dal produttore.

Fortunatamente le versioni personalizzate continuano ad usare i database nativi di Android. Questo implica che un utente può utilizzare più versioni della stessa applicazione continuando a fruire comunque degli stessi dati. Quindi supponendo un dispositivo con la Rubrica personalizzata, è comunque possibile utilizzare la Rubrica implementata per il nostro progetto (vedi par. 4.3).

Gli sviluppi futuri possibili sono svariati. Alcuni possono ricalcare parte dei requisiti del modello originale, come la possibilità di effettuare il controllo dell'accesso anche per Documenti Condivisi.

Inoltre si potrebbe estendere le informazioni di contesto con la posizione fisica dell'utente. Le politiche saranno quindi definite anche in base alla posizione corrente (Location-Aware). Per l'implementazione di tale funzionalità occorre utilizzare la triangolazione GSM e/o il segnale GPS integrato nello smartphone, caratteristiche ampiamente supportate in Android.

Un'ultima considerazione sull'uso dei *Widget* Android per rafforzare la caratteristica *user-friendly* del nostro progetto. Un Widget è un componente di Android che espone informazioni rilevanti di un'applicazione (a colpo d'occhio) direttamente sulla *Home Screen* della piattaforma. In altre parole ha un comportamento simile a quello di una “scorciatoia” nel senso classico dell'ambiente PC desktop. Differisce da quest'ultima perché la sua esecuzione non porta al lancio dell'applicazione ma ad una parte di essa, una parte così rilevante che è utile tenerla “a portata di mano” sulla Home Screen.

Si potrebbe dunque implementare un Widget in grado richiamare una semplice funzionalità che permetta di impostare in modo manuale lo *Status*

dell'utente. Ad esempio, è possibile che un Meeting non venga impostato nel calendario degli eventi, per negligenza o perché indetto all'ultimo minuto. L'uso del Widget, e quindi dell'impostazione manuale, permette di superare questo problema dando la possibilità di continuare ad affidarsi al nostro middleware.

Bibliografia

Abowd, A. K. (2000). Towards a Better Understanding of Context and Context-Awareness. *Georgia Institute of Technology* .

AutoriVari. (2009). OWL 2 web ontology language primer. *W3C Recommendation* .

B Schilit, N. A. (1994). Context-Aware Computing Application. *1st International Workshop on Mobile Computing* .

Chen, H. (2004). An Intelligent Broker Architecture for Pervasive Context-Aware Systems. *PhD Thesis* . University of Maryland, Baltimore County.

CNET. (2010). *Woz riffs on Android vs. iOS*. Tratto da CNET: http://news.cnet.com/8301-13506_3-20023216-17.html

E. Reshetova, J. E. (2008). Mandatory access control for mobile device. *Nokia Research Center Journal* .

Frank. (2010). *Android vs iPhone*. Tratto da <http://stevenf.com/wiki/android%20vs.%20iphone.txt>

Graham. (2010). *Android Shakes Up U.S. Smartphone Market*. Tratto da NPD group: http://www.npd.com/press/releases/press_100510.html

J. I. Hong, J. A. (2001). Running head: context-aware computing infrastructures.

L. Gomez, A. L. (2009). Access Control in Mobile and Ubiquitous Environments. *IGI* .

M. Baldauf, S. D. (2004). A survey on context-aware system. *Int J. Ad Hoc and Ubiquitous Computing Vol. 2 No. 4* .

- Manola, M. (2004). Rdf Primer. *W3C Recommendation* .
- Moses, T. (2005). eXtensible Access Control Markup Language - Specification Document.
- Poslad, S. (2009). *Ubiquitous Computing. Smart devices, environments and.* Wiley.
- Stojanovic, D. (2009). Context-Aware mobile and ubiquitous computing for enhanced usability – Adaptative technologies and applications. *Information science reference* .
- Strang, L.-P. (2004). A Context Modeling Survey, First International Workshop on Advanced Context Modelling, Reasoning and Management. *UbiComp* .
- Toninelli, A., Montanari, R., Kagal, L., & Lassila, a. O. (2007). Proteus: A Semantic Context-Aware Adaptive Policy Model. *IEEE Computer Society Press (2007)* .
- Toninelli, A., Montanari, R., Lassila, O., & D., K. (2009). What's on Users'Minds - Toward a Usable Smart Phone Security Model. *IEEE Pervasive Computing vol. 8, no. 2* .
- Weiser, M. (1991). The computer for the 21st Century.

Ringraziamenti

Un grazie sincero all'infinita pazienza della Professoressa Montanari e la grande disponibilità del Professore Omicini che mi hanno permesso di realizzare questo progetto.

Alla mia famiglia, che mi ha sostenuto durante il mio (a dir poco) lungo percorso universitario.

Alla mia famiglia acquisita, che ha creduto in me nonostante le apparenze.

Agli amici che hanno reso questi anni indimenticabili.

A Nicoletta, la donna della mia vita che mi ha sostenuto, sopportato, spronato, motivato, ispirato e amato facendomi sentire l'uomo più fortunato del mondo.