POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Tema d'anno in Linguaggi e Tecnologie Web

Estensione semantica del progetto Google Physical Web

Docenti del corso:

Prof. Ing. Vincenzo Di Lecce

Ing. Saverio Ieva

Ing. Floriano Scioscia

Studente:

Basile Giorgio

SOMMARIO

Capitolo 1: Physical Web

1.1 Architettura e paradigma generale

Il **Physical Web**[1] è un progetto open source sponsorizzato da Google e aperto al pubblico nel settembre 2014, che si propone di abilitare un interfacciamento immediato, aperto e user-friendly alla moltitudine di smart object che progressivamente stanno prendendo corpo nell'era digitale. **Il progetto nel suo insieme rappresenta un approccio atto a fornire una piattaforma interoperabile per la costruzione del Web of Things, basandosi sulle stesse linee guida che hanno portato allo sviluppo del Web come lo conosciamo oggi, sfruttandone la solidità e l'ormai consolidata architettura**, e avvantaggiandosi di una delle sue caratteristiche principali: l'interazione on-demand. Secondo questa visione, un utente dovrebbe essere in grado di interagire in maniera immediata con ciascuno smart device con un semplice tap sul suo dispositivo portatile, invece di dover obbligatoriamente scaricare un' app dedicata. Infatti**, l'assunzione secondo cui ciascun dispositivo richieda una specifica applicazione** (rilasciata dal proprio produttore) **è semplicemente non realistica**. Il Physical Web non intende però rimpiazzare le app native: si tratta di facilitare l'interazione quando tali app non garantiscono una esperienza fluida e immediata ed abilitare alcuni casi d'uso che sarebbero altrimenti impraticabili:

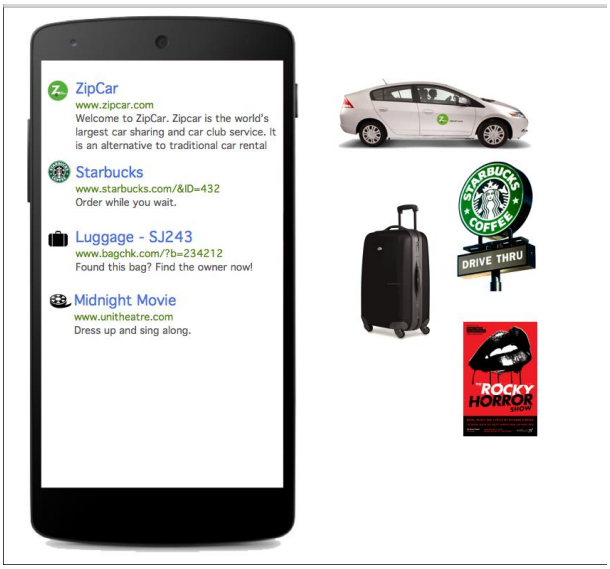


Figura 1 Ciascun dispositivo può fornire informazioni o un'interazione immediata

- il collare di un gatto notifica il numero a cui telefonare per trovare il padrone;
- un autobus comunica la sua prossima fermata;
- ciascun negozio potrà fornire una nuova esperienza digitale una volta entrati;
- una ZipCar invia in broadcast una pagina di registrazione che permetta immediatamente di noleggiarla.

Ciascuno di questi esempi, presi singolarmente, rappresentano un modesto contributo al paradigma generale alla base dell'**Internet of Things**. Presi però nel loro insieme, essi abiliterebbero una *long tail* dove qualunque oggetto o luogo possa offrire informazioni (Figura 1).

Come detto, il Physical Web punta a sfruttare l'architettura del Web così come lo conosciamo oggi, con il fine di **estenderlo verso il mondo fisico** dato dagli smart object intorno a noi. Ciò comporta la creazione di un ecosistema dove ciascun oggetto sia in grado di **inviare in broadcast un URL nell'area che lo circonda**. Ciascun dispositivo personale, come smartphone o tablet, può effettuare il **discovery** di questi URL, e mostrare sul proprio display una lista utilizzabile

[1] https://google.github.io/physical-web/

dall'utente. Questo approccio presenta delle similarità con il comune comportamento di un **motore di ricerca**:

- L'utente richiede una lista di ciò che è intorno a sé
- Una lista ordinata di URL viene mostrata
- L'utente ne sceglie uno
- L'URL viene aperto nella finestra di un browser

Nonostante l'idea di base sia piuttosto semplice, essa genera però una serie di domande molto interessanti:

1. *Bisognerà scaricare un'app? Non è un controsenso?*

Attualmente si parla di un prototipo che permette agli sviluppatori di sperimentare sull'idea generale, ma in futuro potrebbe essere una **funzionalità nativa di ogni sistema operativo**, come avviene con il discovery WiFi, ad esempio. Ad oggi, per le funzionalità di discovery sono state implementate un' app Android, oggetto di questo lavoro, e una per iOS.

2. *Non c'è il rischio di ottenere una lista piuttosto lunga di dispositivi nei paraggi?*

Inizialmente, il numero di dispositivi sarà piuttosto limitato, ma in caso di successo, esso aumenterà sensibilmente e si incorrerà in problemi di user experience. In questo scenario, l'utilizzo di meccanismi di **ranking** si rende necessario. Nei comuni motori di ricerca, siamo soliti scrivere la parola *"tennis"* e ricevere milioni di risultati, confidando che i primi 10 siano i migliori. Allo stesso modo**, lo smartphone potrà essere in grado di ordinare in base sia alla potenza del segnale in ricezione sia in base alle preferenze e allo storico dell'utente, o ancora considerando altri fattori.**

3. *Perché gli URL?*

Il valore degli URL è legato al fatto che essi sono una parte fondamentale del Web, sono molto flessibili e, ancora più importante, decentralizzati. Gli URL permettono a tutti di essere coinvolti, senza alcun tipo di collo di bottiglia. Detto questo, è auspicabile che vengano sperimentati dei modelli del tipo **URL + ID** che vengano utilizzati da server proprietari (es. safeurls.com/?id=12345). Questo è assolutamente consentito e incoraggiato. Tali sistemi saranno in grado di fornire una maggiore sicurezza e un miglior controllo.

4. *Questo approccio è sicuro/privato?*

Il broadcast degli URL avviene "in chiaro", dunque chiunque può vederli, ed è previsto dall'architettura proposta. Per questo motivo, inizialmente è preferibile utilizzare questo paradigma in spazi pubblici, al fine di evitare problemi legati all'utilizzo domestico, ove ad esempio chiunque possa sapere quali oggetti sono presenti nella casa accanto alla sua. D'altra parte, uno dei vantaggi dell'utilizzo di URL sta nel fatto che, data la loro grande flessibilità, esistono diversi modi per incrementarne la sicurezza:

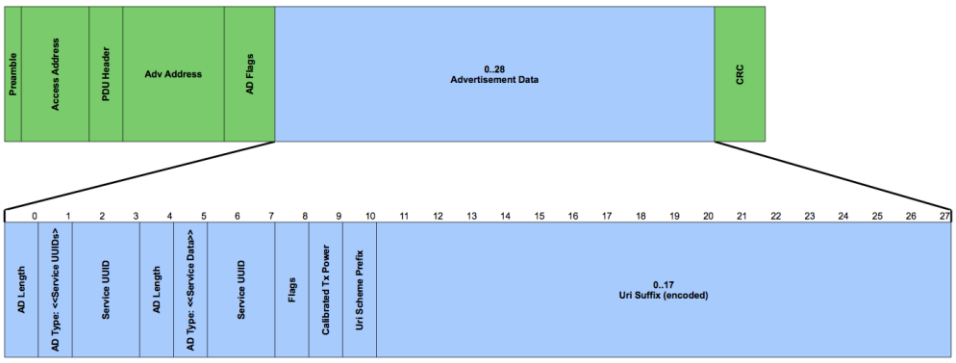
- Una pagina web può prevedere meccanismi di login
- L'URL esposto da un dispositivo può essere periodicamente cambiato

- L'URL può riferirsi ad un indirizzo IP accessibile solo se connessi ad una specifica rete locale

Inoltre, il meccanismo di beaconing passivo fa sì che **l'utente sia completamente invisibile a qualunque beacon, e quindi non può essere tracciato quando esegue un semplice discovery,** ma solo quando decide di aprire una pagina Web.

1.2 UriBeacon

La versione iniziale di questo sistema prevede l'utilizzo di **Bluetooth Low Energy (BLE)** per inviare in broadcast un URL nel BLE Advertising packet, essendo tale standard largamente diffuso su tutti gli smartphone e tablet odierni. Inoltre, esistono piccoli dispositivi BLE che possono effettuare un broadcast continuo **per quasi 2 anni** con una singola batteria a bottone, garantendo dunque una grande efficienza energetica. Lo scopo di ciascun dispositivo è esclusivamente quello di inviare un URL nell'area circostante, che possa essere letto in qualsiasi momento, affrontando in maniera efficiente il *worst case* in cui abbiamo un'area con un gran numero di smart object e ogni utente si connette ad ogni dispositivo, facendo in modo che invece **gli URL vengano passivamente esposti nell'area circostante** senza alcuna frizione e senza che l'utente lasci alcuna traccia del suo passaggio in una certa area.



Offset	Size	Value	Name	Description
0	1	3	AD Length	
1	1	0x03	AD Type	Complete List of 16-Bit Service
2	2	0xFED8	Service ID	Assigned Uri Service UUID
4	1	5..23	AD Length	
5	1	0x16	AD Type	Service Data
6	2	0xFED8	Service ID	Assigned Uri Service UUID
8	1	-	Flags	UriBeacon Flags
9	1	-	TX Power	UriBeacon Tx Power Level
10	1	-	Uri Scheme	UriBeacon Uri Scheme Prefix
11	0..17	octets	Encoded Uri	UriBeacon Uri

Figura 2 La specifica UriBeacon

L'incapsulamento degli URL nei relativi beacon avviene utilizzando il formato **UriBeacon**[2], una specifica rilasciata sotto licenza Apache 2.0 che utilizza i messaggi di Advertising di Bluetooth 4.0 al fine di incapsulare degli URL opportunamente codificati per garantire una buona efficienza. In questa prima fase, è stato definito un **UUID dedicato**, settato al valore custom 0xFED8, che indica che tale beacon rappresenta un UriBeacon (Figura 2). Le dimensioni ridotte dell'ADV packet non permettono di incapsulare URL molto lunghi, consentendo una lunghezza massima di 18 byte.

Questo è ovviamente uno dei compromessi da accettare per evitare connessioni attive con i dispositivi, tenendo però presente che UriBeacon adotta uno schema di encoding che va a comprimere in un singolo carattere alcuni pattern ricorrenti come *http://www.* o *.com*, in maniera molto simile ad altre tecniche usate in NDEF o nei QRCode. Inoltre, l'utilizzo di URL Shortener permette comunque di poter utilizzare URL particolarmente lunghi. E' importante

[2] https://github.com/google/uribeacon

notare come la specifica sia aperta all'utilizzo di **qualsiasi tipo di URI (e dunque sia URL che URN)**: questa generalità è fondamentale se pensiamo all'approccio alla base del paradigma **Linked Open Data** e del **Semantic Web**, oggetti di questo lavoro, che permettono di definire mediante URI qualsiasi tipo di concetto, luogo o entità, non limitandosi dunque a puntare pagine HTML.

PROPRIETARIE	OPEN SOURCE
Blesh	**Android**
KST	OpenBeacon
Accent Systems	Arduino
iBlio	**NodeJS**
BKON	tessel

Figura 3 Piattaforme che supportano UriBeacon

Attualmente, diversi produttori di dispositivi di beaconing forniscono alcuni apparecchi che supportano la specifica UriBeacon, rendedoli facilmente configurabili da remoto. In alternativa, è possibile utilizzare dei firmware per piattaforme specifiche, che consentono di esporre URI secondo la medesima specifica in maniera completamente gratuita e open source (Figura 3).

In alternativa ad UriBeacon, per casi d'uso riguardanti il discovery di oggetti domestici, sono previste altre modalità di esposizione mediante mDNS ed SSDP, che non sono state però oggetto di studio e utilizzo nel presente lavoro.

1.3 Discovery e proxy di cache per il recupero di metadati

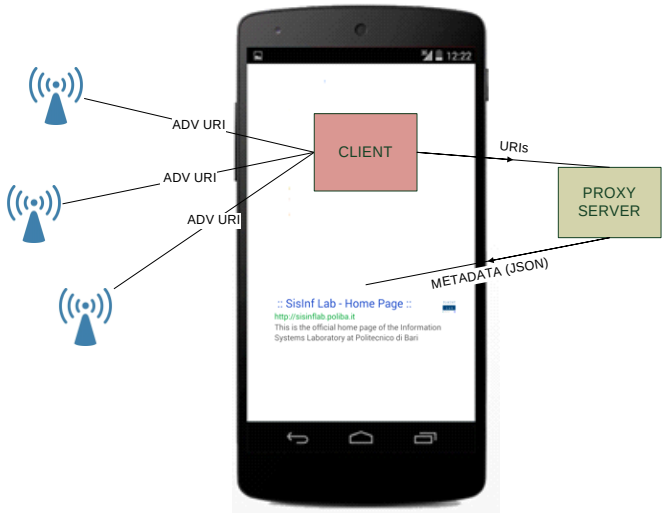


Figura 4 Procedura di discovery

Il discovery dei beacon presenti in una certa area viene attualmente effettuato mediante un client dedicato, sviluppato per le piattaforme Android e iOS. All'apertura dell'app, viene mostrata una lista dei beacon visibili, ordinata in base alla potenza del segnale in ricezione. Tale metrica è ovviamente influenzabile da molteplici fattori, ma nella pratica funziona piuttosto bene. Questo è il motivo per il quale nell'ADV packet viene incluso un campo TX_POWER, al fine di calcolare la discrepanza rispetto alla potenza trasmissiva e permettere dunque un ranking più efficace.

I client vanno dunque a listare, per ciascun URL, i **metadati** della pagina Web a cui si riferiscono: **titolo, descrizione, URL e favicon**. Tali metadati potrebbero essere recuperati ogni volta a run-time ma, per ridurre le latenze e il consumo dati, è previsto l'utilizzo di un **proxy server di cache** (Figura 4Figura 5), all'indirizzo *http://url-caster.appspot.com/resolve-scan*[3]. Il client invia una richiesta **POST** con entity body dato dalla lista di URL trovati (formattati in **JSON**) e questo server risponde con una struttura **JSON** contenente i metadati di ognuno (Figura 5).

[3] Disponibile anche con interfaccia web all'indirizzo http://url-caster.appspot.com/webui

Figura 5 Dopo aver richiesto i metadati di 4 pagine HTML, il client legge la risposta JSON e mostra i risultati

Il prototipo del client Android invia una richiesta di risoluzione dei metadati per ogni beacon trovato, sebbene il proxy supporti risoluzioni multiple. Ovviamente l'approccio va migliorato ma in questa prima fase di sviluppo fornisce ugualmente buone prestazioni. Si noti che il proxy server non è strettamente necessario ma ha un buon impatto prestazionale sul sistema, anche perché in grado di effettuare la **risoluzione di short URL**, non richiedendo dunque ulteriori elaborazioni lato client.

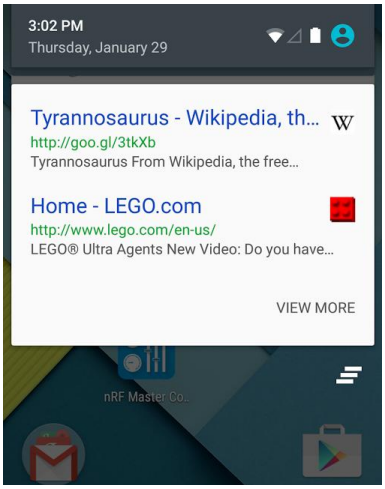


Figura 6 Risultati discovery nel pannello delle notifiche

Quando l'utente non utilizza l'app, in background viene eseguito periodicamente un servizio di discovery che va a mostrare una notifica non invasiva (Figura 6), che mostra il numero di UriBeacon rilevati e i primi due beacon della lista.

I metadati recuperati attraverso il proxy sono dunque relativi alle pagine Web puntate dagli URL. Il sistema si aspetta che tali URL si riferiscano quindi a pagine HTML valide. Il problema delle pagine web mancanti è gestito secondo il tradizionale approccio del web *404 Not Found*, ma ovviamente in futuro sarà necessario prevedere un meccanismo di recupero di metadati che prescinda dall'utilizzo di pagine HTML, **nel caso in cui gli URL vengano usati come link per applicazioni native**.

Ovviamente, in questo grande scenario interattivo, un ruolo fondamentale è giocato dalla capacità di ranking delle risorse disponibili. L'utilizzo del proxy server o un arricchimento delle capacità del client possono sicuramente aiutare a migliorare questo task in 2 modi:

- Tenendo traccia degli URL più cliccati in modo che URL molto richiesti abbiano un rank più alto
- Tracciando informazioni personali in modo che se, ad esempio, quando un utente è a lavoro è solito utilizzare un particolare URL, anch'esso venga marcato con un rank maggiore.

In Figura 7 viene mostrato il codice Java per la composizione di richieste JSON e relativi metodi callback con la libreria Android Volley.

```
private static JSONObject createUrlMetadataRequestObject(String url, int txPower, int rssi) {
  JSONObject jsonObject = new JSONObject();
  try {
    JSONArray urlJsonArray = new JSONArray();
    JSONObject urlJsonObject = new JSONObject();
    urlJsonObject.put("url", url);
    urlJsonObject.put("txpower", txPower);
    urlJsonObject.put("rssi", rssi);
    urlJsonArray.put(urlJsonObject);
    jsonObject.put("objects", urlJsonArray);
  } catch (JSONException ex) {
    Log.d(TAG, "error: " + ex);
  }
  return jsonObject;
}

private static JsonObjectRequest createUrlMetadataRequest(JSONObject jsonObj, final boolean isDemoRequest) {
  return new JsonObjectRequest(
      isDemoRequest ? DEMO_METADATA_URL : METADATA_URL,
      jsonObj,
      new Response.Listener<JSONObject>() {
        // Called when the server returns a response
        @Override
        public void onResponse(JSONObject jsonResponse) {

          // Build the metadata from the response
          try {
            JSONArray foundMetaData = jsonResponse.getJSONArray("metadata");

            // Loop through the metadata for each url
            if (foundMetaData.length() > 0) {

              for (int i = 0; i < foundMetaData.length(); i++) {

                JSONObject jsonUrlMetadata = foundMetaData.getJSONObject(i);

                String title = "";
                String url = "";
                String description = "";
                String iconUrl = "/favicon.ico";
                String id = jsonUrlMetadata.getString("id");
                float score = UNDEFINED_SCORE;

                if (jsonUrlMetadata.has("title")) {
                  title = jsonUrlMetadata.getString("title");
                }
                if (jsonUrlMetadata.has("url")) {
                  url = jsonUrlMetadata.getString("url");
                }
                if (jsonUrlMetadata.has("description")) {
                  description = jsonUrlMetadata.getString("description");
                }
                if (jsonUrlMetadata.has("icon")) {
                  // We might need to do some magic here.
                  iconUrl = jsonUrlMetadata.getString("icon");
                }
                if (jsonUrlMetadata.has("score")) {
                  score = Float.parseFloat(jsonUrlMetadata.getString("score"));
                }

                // TODO: Eliminate this fallback since we expect the server to always return an icon.
                // Provisions for a favicon specified as a relative URL.
                if (!iconUrl.startsWith("http")) {
                  // Lets just assume we are dealing with a relative path.
                  Uri fullUri = Uri.parse(url);
                  Uri.Builder builder = fullUri.buildUpon();
                  // Append the default favicon path to the URL.
                  builder.path(iconUrl);
                  iconUrl = builder.toString();
                }

                // Create the metadata object
                UrlMetadata urlMetadata = new UrlMetadata();
                urlMetadata.title = title;
                urlMetadata.description = description;
                urlMetadata.siteUrl = url;
                urlMetadata.iconUrl = iconUrl;

                // Kick off the icon download
                downloadIcon(urlMetadata);

                if (isDemoRequest) {
                  mMetadataResolverCallback.onDemoUrlMetadataReceived(id, urlMetadata);
                } else {
                  mMetadataResolverCallback.onUrlMetadataReceived(id, urlMetadata);
                }
              }

            }
          } catch (JSONException e) {
            e.printStackTrace();
          }
        }
```

Figura 7 Richiesta JSON e callback di rispostaper il recupero di metadati al proxy server

Capitolo 2: Estensione semantica proposta

2.1 Discovery e ranking di risorse su base semantica

Nella visione fin qui analizzata, un ruolo significativo potrebbe essere rappresentato **dell'esposizione di descrizioni delle risorse disponibili** in una determinata area. Nella documentazione del progetto[4], si fa proprio riferimento alla possibilità di prevedere che gli URL inviati in broadcast non si riferiscano solo a pagine HTML valide, ma in alcuni casi può essere interessante che un oggetto possano esporre dei metadati, senza che questi dipendano da una pagina web nello specifico.

In questo scenario, la possibilità di utilizzare dati semantici per il discovery e il ranking di risorse nel Physical Web si propone come una possibilità concreta di estensione dello stato dell'arte, sfruttando le tecnologie alla base del **Semantic Web**. In particolare, l'architettura proposta potrebbe consentire di **incapsulare delle annotazioni semantiche nei beacon**, per descrivere le caratteristiche e le funzionalità offerte dagli oggetti. In questo modo sarebbe possibile non solo estendere gli scenari prima analizzati, ma anche prevedere delle interazioni **machine-to-machine**.

Considerando che il framework si basa sull'esposizione di un URI, che garantisce una certa universalità di utilizzo, come è possibile utilizzare la semantica?

Una soluzione ragionevole di prevedere che l'URI esposto da un beacon punti ad un documento, contenente un'annotazione semantica, che sia presente sul web con un certo indirizzo (es. *http://sisinflab.poliba.it/device1.owl)* da inviare in broadcast tramite UriBeacon. In questo modo, si abiliterebbero tre possibili modalità di utilizzo:

1. **l'URL punta direttamente ad una pagina HTML** (scenario standard) e l'utente cliccandovi può accedere al documento tramite browser;
2. **l'URL punta ad un'annotazione semantica,** al fine di abilitare funzionalità M2M e di discovery su base semantica; l'utente o un agente possono comporre una richiesta su base semantica e fare un matchmaking, ottenendo uno score, utilizzabile per il ranking delle risorse, che fornisce una misura di quanto una certa risorsa soddisfa la richiesta dell'utente;
3. **l'URL punta ad una pagina HTML annotata semanticamente**; l'utente può sia visualizzare la pagina HTML su browser, sia utilizzare le funzionalità legate alla semantica.

La soluzione proposta (Figura 8) prevede quindi l'introduzione di un **layer semantico** [1][2] in grado di abilitare funzionalità di discovery e ranking, che necessita ovviamente dell'utilizzo di un ragionatore. L'idea è quella di integrare il ragionatore **Mini-ME**[3] nell'applicazione client, in modo che una volta raccolte le annotazioni semantiche relative a tutti i beacon rilevati, esse vengano ordinate in base ad uno **score** e restituite all'utente (o all'agente). In questo modo, la natura del framework basata su URI rimarrebbe intatta, abilitando però l'utilizzo di tecniche di inferenza per il ranking delle risorse disponibili.

[4] https://github.com/google/physical-web/blob/master/documentation/technical_overview.md#meta-data

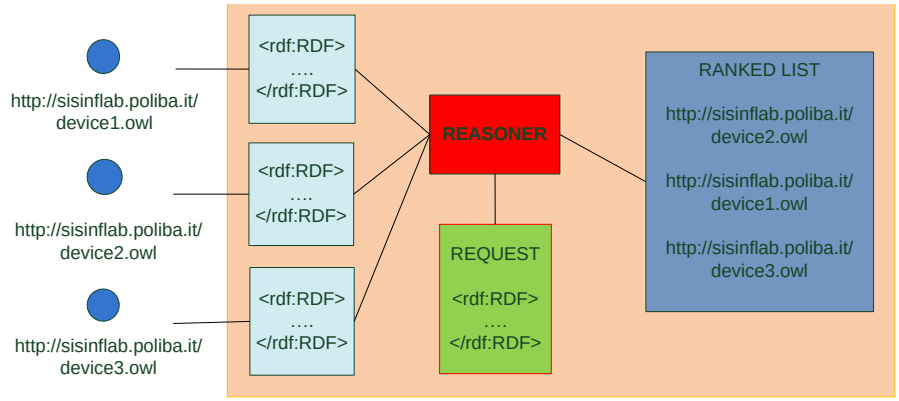


Figura 8 Estensione semantica proposta

L'integrazione del ragionatore all'interno dell'app Android risulta piuttosto naturale, utilizzando entrambi il linguaggio Java, e necessiterebbe in uno scenario reale di un'interfaccia di composizione della richiesta (sia manuale mediante GUI, sia automatica mediante un profiler [4] o un compositore vocale), da inviare al reasoner per confrontarla con le risorse (beacon). I servizi di inferenza non-standard di **Abduzione** e di **Contrazione** [5] sono quelli necessari al calcolo dello score semantico, ma potrebbe essere utilizzata anche l'operazione di **Covering** per fornire un set di oggetti i cui servizi offerti possano essere orchestrati per svolgere un task complesso.

L'utente comporrà una richiesta, rappresentata da un documento che dichiara un'ontologia senza nome, e che importa a sua volta l'URL dell'ontologia che si trova sul web[5] attraverso il costrutto **owl:imports**. In questo modo, è possibile:

- **specificare le ontologie di riferimento senza sapere quali siano a priori**;
- effettuare in automatico il download delle stesse attraverso le OWL API: il metodo manager.loadOntologyFromOntologyDocument() esegue questo task in maniera autonoma, l'unico accorgimento da adottare è, in Android, di eseguire il parsing al di fuori dello UI Thread (al fine di evitare una android.os.NetworkOnMainThreadException);
- **distinguere i beacon che si riferiscono ad ontologie diverse** in modo da assegnarvi un rank "undefined", che li ponga in coda alla lista risultante.

```
<owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://sisinflab.poliba.it/ieva/PhysicalWeb/shopping.owl"/>
</owl:Ontology>
<owl:AnnotationProperty rdf:about="http://sisinflab.poliba.it/ieva/PhysicalWeb/shopping.owl#url"/>

<rdf:Description rdf:about="http://sisinflab.poliba.it/ieva/PhysicalWeb/shopping.owl#RossoPomodoro">
    <url>http://web.rossopomodoro.it/italiano/default.aspx</url>

    <!-- BODY -->

<rdf:Description>
```

Figura 9 Sintassi per l'uso di owl:imports e owl:AnnotationProperty

Per quanto riguarda gli individui esposti tramite UriBeacon, anch'essi presenteranno l'import delle ontologie di riferimento: se nessuna di esse è utilizzata dalla request, il beacon verrà segnato con valore di ranking molto alto che ne denoterà la "non pertinenza". Nel caso di corrispondenza, il reasoner effettuerà il parsing del documento, **senza andare però ad instanziare nuove ontologie in memoria**, in quanto le OWL API, utilizzate da Mini-ME, sono in grado di riconoscere la corrispondenza degli URI, e di capire che una data ontologia è stata già caricata. Inoltre, prima della descrizione vera e propria, viene dichiarata una **owl:AnnotationProperty**, che verrà utilizzata per specificare l'URL di una eventuale pagina HTML verso cui indirizzare l'utente in caso di tap (Figura 9).

[5] http://protegewiki.stanford.edu/wiki/How_Owl_Imports_Work

2.2 UriBeacon per l'esposizione di annotazioni RDF/OWL

```
public class UriBeaconAdvertiserActivity extends Activity {

    private static final String TAG = "UriBeaconAdvertiser";
    private static final int ENABLE_BLUETOOTH_REQUEST = 17;
    // Really 0xFED8, but Android seems to prefer the expanded 128-bit UUID version
    private static final ParcelUuid URI_BEACON_UUID = ParcelUuid.fromString("0000FED8-0000-1000-8000-00805F9B34FB");

    private BluetoothAdapter bluetoothAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_uri_beacon);

        setupBluetooth();
    }

    private void advertiseUriBeacon() {

        BluetoothLeAdvertiser bluetoothLeAdvertiser = bluetoothAdapter.getBluetoothLeAdvertiser();

        AdvertiseData advertisementData = getAdvertisementData();
        AdvertiseSettings advertiseSettings = getAdvertiseSettings();

        bluetoothLeAdvertiser.startAdvertising(advertiseSettings, advertisementData, advertiseCallback);
    }

    private AdvertiseData getAdvertisementData() {
        AdvertiseData.Builder builder = new AdvertiseData.Builder();
        builder.setIncludeTxPowerLevel(false); // reserve advertising space for URI

        byte[] beaconData = new byte[16];
        beaconData[0] = 0x00; // flags
        beaconData[1] = (byte) 0xBA; // transmit power
        beaconData[2] = 0x03; // https://
        beaconData[3] = 0x67; // g
        beaconData[4] = 0x6F; // o
        beaconData[5] = 0x6F; // o
        beaconData[6] = 0x2E; // .
        beaconData[7] = 0x67; // g
        beaconData[8] = 0x6C; // l
        beaconData[9] = 0x2F; // /
        beaconData[10] = 0x4E;// mcdonalds: http://goo.gl/NzKCu3 --> 4E-7A-4B-43-75-33
        beaconData[11] = 0x7A;//
        beaconData[12] = 0x4B;//
        beaconData[13] = 0x43;//
        beaconData[14] = 0x75;//
        beaconData[15] = 0x33;//

        builder.addServiceData(URI_BEACON_UUID, beaconData);

        // Adding 0xFED8 to the "Service Complete List UUID 16" (0x3) for iOS compatibility
        builder.addServiceUuid(URI_BEACON_UUID);

        return builder.build();
    }
```

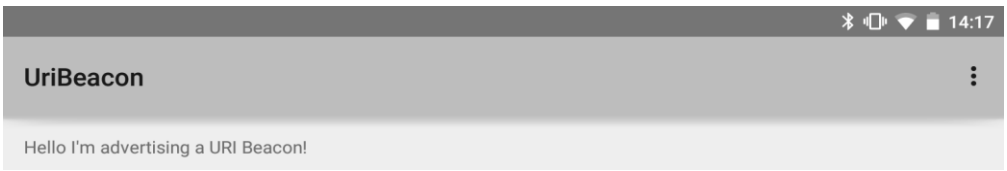


Figura 10 Esposizione di UriBeacon su Nexus 9

Nel presente lavoro, si è reso necessario l'utilizzo di alcuni dei framework open source, elencati nel Capitolo 1, per l'esposizione di UriBeacon. Considerando le apparecchiature disponibili, i target hanno riguardato la piattaforma **Android** e quella **Node.js**[6].

Per quanto riguarda **Android**, attualmente il supporto riguarda tutti i dispositivi dotati **nativamente** di una versione pari o superiore alla 5.0.1 Lollipop, escludendo dunque alcuni dispositivi molto popolari come Nexus 4 e Nexus 5, a causa di alcune limitazioni hardware[7]. Negli scenari testati è stato dunque utilizzato un Nexus 9, installando un'app (Figura 10) dedicata all'esposizione di un beacon. Il codice sorgente è stato modificato in modo da encodare byte per byte i codici esadecimali dei caratteri degli URL.

[6] https://nodejs.org/
[7] https://code.google.com/p/android-developer-preview/issues/detail?id=1570

Inoltre, il presente lavoro ha previsto l'esposizione di beacon in ambiente Linux, mediante l'utilizzo di **Node.js,** un popolare framework per realizzare applicazioni Web in **JavaScript**, permettendoci di utilizzare questo linguaggio, tipicamente utilizzato nella "client-side", anche per la scrittura di applicazioni "server-side". La piattaforma è basata sul **JavaScript Engine V8**, sviluppato da Google, utilizzato anche da Chrome e disponibile sulle principali piattaforme, anche se maggiormente performante su sistemi operativi UNIX-like. Tra questi, rientra il popolare **Raspberry Pi** (Figura 11), dotato di sistema operativo Raspbian (derivato dal più popolare Debian), le cui schede richiedono semplicemente l'installazione di una comune chiavetta Bluetooth 4.0.

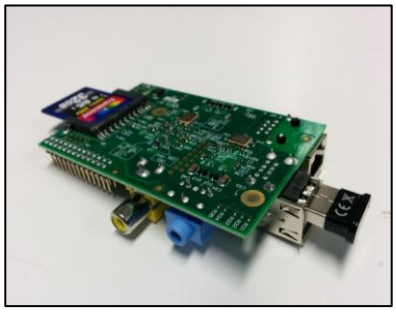


Figura 11 Raspberry Pi con modulo Bluetooth 4.0

La caratteristica principale di Node.js risiede nella possibilità di accedere alle risorse del sistema operativo in modalità **event-driven**, grazie ad **un sistema di callback** gestito a basso livello dall'engine, fornendo una maggiore efficienza in ambito networking.

Inoltre, attraverso il **Node Package Manager** (**npm**) è possibile aggiungere il supporto per numerosi tipi di hardware, funzionalità, estensioni. Installando i **package uri-beacon** e **bleno**, è possibile utilizzare delle procedure di composizione di UriBeacon rispettosi dello standard (Figura 12).

```
var encode = require('./uriEncoder').encode;

var template = new Buffer(10); // maximum 31 bytes
template[0] = 0x03; // Length
template[1] = 0x03; // Parameter: Service List
template[2] = 0xD8; // URI Beacon ID
template[3] = 0xFE; // URI Beacon ID
template[4] = 0x00; // Length
template[5] = 0x16; // Service Data
template[6] = 0xD8; // URI Beacon ID
template[7] = 0xFE; // URI Beacon ID
template[8] = 0x00; // Flags
template[9] = 0x20; // Power

// This won't work for people the want to advertise additional services
var makeBuffer = function (uri) {
    // encode the uri
    var encoded = encode(uri);

    var data = Buffer.concat([template, encoded], template.length + encoded.length);

    // set the length
    data[4] = encoded.length + 5;

    return data;
};
var advertise = function(uri) {

    // scan data is optional, OK to pass new Buffer(0);
    var scanData = new Buffer(createScanData("URI Beacon")); // maximum 31 bytes

    var advertisementData = advertisement.makeBuffer(uri);

    if (advertisementData.length > max_advertisement_length) {
        throw "Encoded URI must be less than " + max_uri_length +
            " bytes. It is currently " + advertisementData.length + " bytes.";
    }

    bleno.startAdvertisingWithEIRData(advertisementData, scanData);

};

var uriBeacon = require('uri-beacon');
uriBeacon.advertise("http://goo.gl/pmgB1g"); //zara
```

Figura 12 Implementazione Javascript di UriBeacon su node.js

2.4 Calcolo dello score

Un ruolo fondamentale nell'ordinamento dei beacon rilevati è ovviamente quello della funzione di score semantico, in base alla richiesta specificata. In questo lavoro, sono stati utilizzati i servizi di Abduzione e Contrazione [4] per ottenere un valore percentuale, che misura quanto la risorsa esposta tramite UriBeacon soddisfa la richiesta. Il calcolo dello score viene effettuato mediante il seguente algoritmo:

1. `boolean comp = checkCompatibility(res, req)` → verifica che non ci siano contrasti tra resource e request, sia a livello di concetti disgiunti sia a livello di restrizioni numeriche;
2. `if (comp == true)`
   a. $score = \left[1 - \left(\frac{abduction(res,req)}{abduction(TOP,req)}\right)\right] * 100$
3. `else`
   a. $score = \left[1 - \left(\frac{contraction(res,req)}{abduction(TOP,req)}\right) + \left(\frac{abduction(res,keep)}{abduction(TOP,keep)}\right)\right] * 100$

Nel caso di **compatibilità**, lo score è tanto più alto quanto la penalty risultante dall'abduzione è bassa, ossia tanto più la richiesta viene coperta dalla risorsa proposta. Nel caso invece di incompatibilità, viene prima utilizzato il servizio di contrazione che, oltre a restituire una certa penalty, fornisce anche la cosiddetta ***Keep***, contenente la parte di request non in contrasto con la resource; essa viene poi utilizzata per una successiva operazione di abduzione con la richiesta. Si noti che tutte le penalty ottenute vengono **normalizzate** rispetto all'abduzione avente come risorsa il concetto TOP, che ci fornisce in sostanza la **penalty massima** ottenibile, in base all' ontologia utilizzata.

Figura 13 Esempi di URL aggiunti ai preferiti (azzurro) e marcati come spam (rosso)

Inoltre, in questo lavoro è stato implementato un meccanismo di base di raccolta delle preferenze dell'utente, implementando le seguenti funzionalità (Figura 13):

- **Cronologia:** memorizza uno storico degli URL delle pagine HTML puntate dai beacon, che sono state aperte dall'utente in un browser;
- **Preferiti:** l'utente può aggiungere un segnalibro su un certo URL;
- **Spam:** l'utente può marcare un URL come spam.

Utilizzando questo meccanismo, è possibile modificare lo score in base ai dati dell'utente; in particolare, per ogni URL:

a) Se è presente tra i **preferiti**, lo score aumenta del 10%;
b) Se è stato marcato come **spam**, lo score viene decrementato del 50%;

Se a) e b) non sono vere, allora viene presa in considerazione la **cronologia**, aumentando lo score del 5% ogni 3 click registrati (fino ad un massimo del 15%, ossia dopo la rilevazione di 9 click lo score non viene ulteriormente incrementato). In Figura 14 è mostrato il codice Java per il calcolo dello score, che tiene conto di tutti gli

aspetti fin qui esaminati.

```
public double calculateScore(IRI supply, IRI demand, String refUrl, UserData data) {

    double score;

    if (reasoner.checkCompatibility(supply, demand)) {
        //Log.i(TAG, "Compatibility");
        Abduction abd = abduction(supply, demand);
        //Log.i(TAG, "Abd penalty " + String.valueOf(abd.penalty));

        Item top = new Item(IRI.create("http://www.w3.org/2002/07/owl#Thing"));
        IRI topIRI = reasoner.loadSupply(top);
        Abduction max = abduction(topIRI, demand);
        //Log.i(TAG, "Max penalty " + String.valueOf(max.penalty));
        score = (1 - (abd.penalty / max.penalty)) * 100;

    } else {
        //Log.i(TAG, "Incompatibility");
        Contraction cnt = contraction(supply, demand);
        //Log.i(TAG, "Contr penalty " + String.valueOf(cnt.penalty));

        Item keep = new Item(IRI.create("Keep" + keepCount), cnt.K);
        keepCount++;
        IRI keepIRI = reasoner.loadDemand(keep);

        Abduction abd = abduction(supply, keepIRI);
        //Log.i(TAG, "Abd penalty " + String.valueOf(abd.penalty));

        Item top = new Item(IRI.create("http://www.w3.org/2002/07/owl#Thing"));
        IRI topIRI = reasoner.loadSupply(top);
        Abduction max = abduction(topIRI, keepIRI);

        //Log.i(TAG, "TOTAL penalty " + String.valueOf(max.penalty));

        score = (1 - ((abd.penalty / max.penalty) + (cnt.penalty / max.penalty))) * 100;


    }
    if(data.getBookmarks().contains(refUrl)){
        //se è un segnalibro, aumenta del 10%
        score += 10;
    }else if(data.getSpam().contains(refUrl)){
        //se è spam, diminuisci del 50%
        score -= 50;
    }else{
        int count = 0;
        for(String url : data.getHistory()){
            if(url.equalsIgnoreCase(refUrl))
                count++;
        }
        //ogni 3 click, aumenta del 5% per un massimo del 20%
        int plus = (Math.round(count/3)) * 5;
        if(plus > 15)
            plus = 15;
        score += plus;
    }

    if(score > 100)
        score = 100;
    else if(score < 0){
        score = 0;
    }
    return score;
}
```

Figura 14 Calcolo score in Java su piattaforma Android

2.5 Workflow del client Android e misura di performance

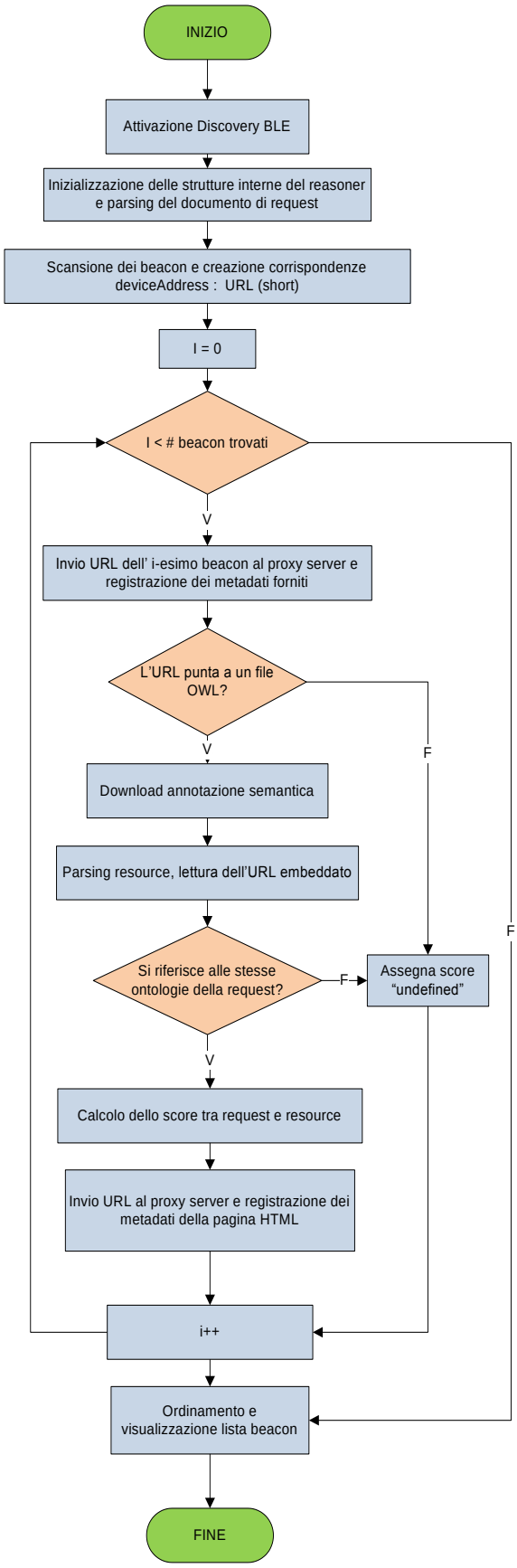


Figura 15 Workflow del discovery** su **base semantica

In Figura 15, è rappresentato l'intero workflow della procedura di discovery arricchita per via semantica. Per ogni beacon rilevato, essa prevede il download sia dei metadati di due URL (annotazione esposta e URL specificato al suo interno), sia quello dell'annotazione semantica puntata dal beacon. Questa estensione ovviamente comporta un sensibile aumento del traffico di rete per il dispositivo, oltre che del turnaround time complessivo.

Per valutare questi aspetti, il client è in grado di misurare i tempi di completamento di tutti i singoli step previsti dall'algoritmo per ciascuno dei beacon, ed è in grado di fornire delle statistiche di traffico in entrata e in uscita dal momento dell'accensione del telefono, utilizzando la classe TrafficStats di Android. Grazie al proxy server di cache, l'impatto sul **turnaround time** risulta essere piuttosto limitato, in quanto mediamente l'intera fase di discovery per il singolo beacon è pari a circa **0.8 secondi**, variabile a seconda dei tempi di risposta della rete, tempo accettabile dal punto di vista della UX, considerando che ogni beacon esegue un thread non bloccante.

Il **traffico di rete** per ciascun beacon risulta essere invece di circa **5 KiB in entrata e 2 KiB in uscita**. Considerando che il dispositivo utilizza un piano dati limitato mensilmente, la riduzione di questi valori porterebbe sicuramente un beneficio. A tal proposito, la community sta provvedendo ad eliminare l'overhead dovuto al download delle icone che attualmente viene effettuata con una richiesta http del client, aggiuntiva rispetto al recupero di metadati dal proxy. Inoltre, potrebbe essere opportuno **evitare di scaricare i metadati per annotazioni che forniscono uno score al di sotto di una certa soglia** e dunque ritenuti non pertinenti rispetto alla richiesta effettuata, restringendo di molto il numero di beacon per il quale è necessaria un'ulteriore interazione con il proxy. Si noti che tutte le richieste sono gestite in

maniera asincrona, con un sistema di metodi callback che vanno poi ad implementare il workflow proposto (Figura 16).

```
@Override
public void onAnnotationDownloaded(final String url, final String annotation) {
    Performances perf = mUrlToPerformances.get(url);
    perf.stopTimer(Performances.SEMANTIC_TIME_INDEX);
    perf.addDownData(annotation.getBytes().length);
    perf.startTimer(Performances.SCORE_TIME_INDEX);
    new AsyncTask<Void, Void, Void>() {
        HashMap<String, Object> supData = null;

        @Override
        protected Void doInBackground(Void... params) {
            try {
                InputStream is = new ByteArrayInputStream(annotation.getBytes());
                supData = semanticReasoner.loadResource(is);
            } catch (OWLOntologyCreationException e1) {
                e1.printStackTrace();
            }
            return null;

        }

        @Override
        protected void onPostExecute(Void aVoid) {
            super.onPostExecute(aVoid);
            Performances perf = mUrlToPerformances.get(url);

            IRI supply = (IRI) supData.get("iri");
            if (supply != null && semanticReasoner.getDemand() != null) {
                double score = semanticReasoner.calculateScore(supply, semanticReasoner.getDemand(), (String) supData.get("url"), userData);
                perf.stopTimer(Performances.SCORE_TIME_INDEX);
                SemanticData d = new SemanticData(annotation, score);
                d.setIconUrl((String) supData.get("icon"));
                d.setRefUrl((String) supData.get("url"));
                mUrlToSemantic.put(url, d);
                if (d.getRefUrl() != null) {
                    perf.startTimer(Performances.SITE_TIME_INDEX);

                    int bytes = InnerMetadataResolver.findInnerUrlMetadata(getActivity(), NearbySemanticBeaconsFragment.this, url, d.getRefUrl(), 0, 0);
                    perf.addUpData(bytes);
                }
            } else {
                SemanticData d = new SemanticData(annotation, SemanticReasoner.UNDEFINED_SCORE);
                d.setIconUrl((String) supData.get("icon"));
                d.setRefUrl((String) supData.get("url"));
                mUrlToSemantic.put(url, d);
                if (d.getRefUrl() != null) {
                    perf.startTimer(Performances.SITE_TIME_INDEX);

                    int bytes = InnerMetadataResolver.findInnerUrlMetadata(getActivity(), NearbySemanticBeaconsFragment.this, url, d.getRefUrl(), 0, 0);
                    perf.addUpData(bytes);
                }
            }
            mNearbyDeviceAdapter.sortDevices();
            mNearbyDeviceAdapter.notifyDataSetChanged();
        }
    }.execute();
}

@Override
public void onUrlMetadataReceived(String url, MetadataResolver.UrlMetadata urlMetadata) {

    mUrlToUrlMetadata.put(url, urlMetadata);

    Performances perf = mUrlToPerformances.get(url);
    perf.stopTimer(Performances.EXPOSED_TIME_INDEX);
    perf.addDownData(urlMetadata.toString().getBytes().length);
    mNearbyDeviceAdapter.notifyDataSetChanged();

    perf.startTimer(Performances.SEMANTIC_TIME_INDEX);
    AnnotationDownloader.downloadAnnotation(getActivity(), NearbySemanticBeaconsFragment.this, url, urlMetadata.siteUrl);
}

@Override
public void onInnerUrlMetadataReceived(String annUrl, String url, InnerMetadataResolver.UrlMetadata urlMetadata) {
    Performances perf = mUrlToPerformances.get(annUrl);
    perf.stopTimer(Performances.SITE_TIME_INDEX);
    perf.stopTimer(Performances.FULL_TIME_INDEX);
    Log.i(TAG, "Turnaround stop");
    perf.startTimer(Performances.ICON_TIME_INDEX);
    perf.addDownData(urlMetadata.toString().getBytes().length);

    mUrlToSemantic.get(annUrl).setRefUrlMetadata(urlMetadata);
    mNearbyDeviceAdapter.notifyDataSetChanged();
}
```

Figura 16 Callback per la gestione asincrona su Android

Capitolo 3: Casi di studio

I casi di studio analizzati in questo capitolo mostrano in concreto il funzionamento del framework. In particolare, tutte le richieste e le risorse utilizzate si riferiscono all'ontologia http://sisinflab.poliba.it/ieva/PhysicalWeb/shopping.owl, che modella ad un livello non troppo alto le entità dello scenario "ipermercato", prendendo come base di partenza il diagramma[8] di Figura 17. A causa della limitata disponibilità di dispositivi BLE, gli scenari analizzati prevedono sempre il discovery di 3 beacon.

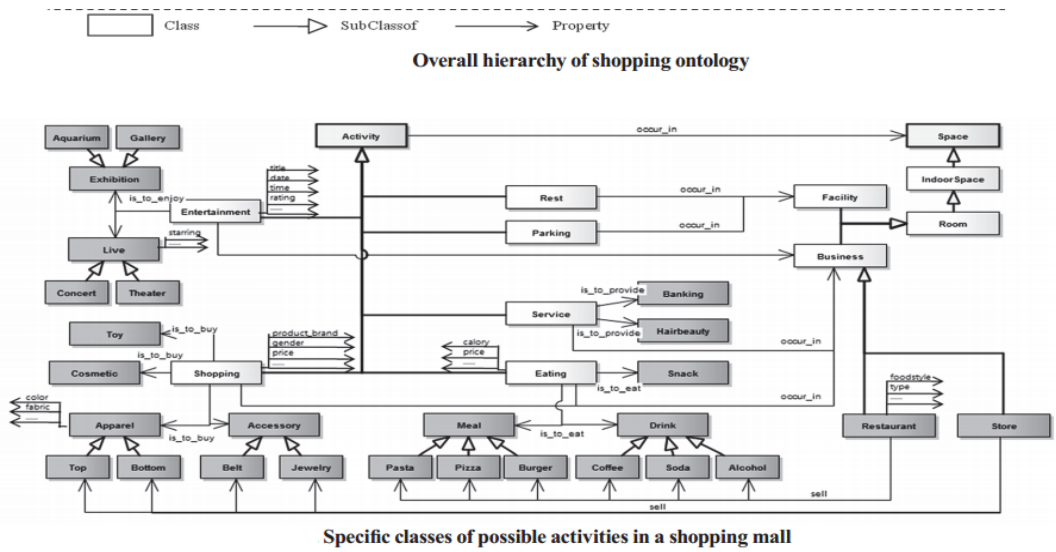


Figura 17 Ontologia dello scenario "ipermercato"

3.1 Scenario 1

In questo primo scenario l'utente sia alla ricerca di **un negozio di scarpe maschili**. Si tratta di una richiesta piuttosto precisa, che va a considerare un concetto molto annidato nell'ontologia che stiamo considerando; intorno all'utente sono presenti 3 beacon:

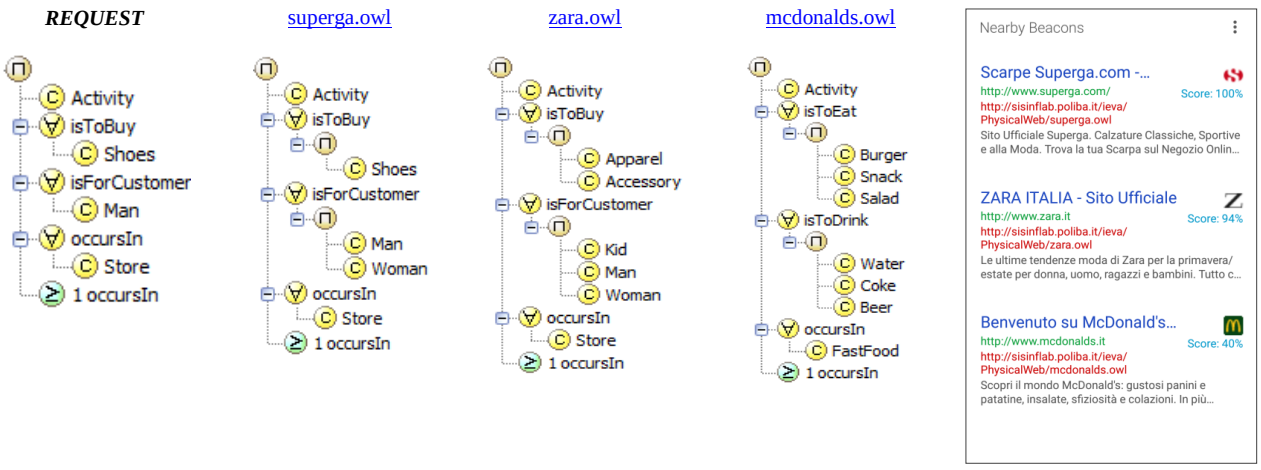


Figura 18 Risultati Scenario 1

[8] http://www.researchgate.net/publication/264170259_Topological_Analysis_in_Indoor_Shopping_Mall_using_Ontology

I risultati di Figura 18 mostrano che l'UriBeacon relativo al negozio Superga ottiene uno score molto alto a seguito del fatto che soddisfa tutti i requisiti della request. Il negozio Zara ottiene uno score leggermente più basso, essendo il concetto *Apparel* meno specifico all'interno dell'ontologia. Infine, la resource McDonald's ottiene uno score molto basso data la non pertinenza rispetto alla richiesta effettuata. Alternativamente, Zara potrebbe anche specificare tutti i concetti sottoclassi di *Apparel* in vendita, tra i quali sarebbe compreso *Shoes*, in tal caso avremmo un match esatto per entrambe le risorse.

3.2 Scenario 2

Il secondo scenario proposto riguarda invece la ricerca di un punto di ristoro. In particolare, l'utente richiede alcuni specifici cibi e bevande, all'interno di un fast food.

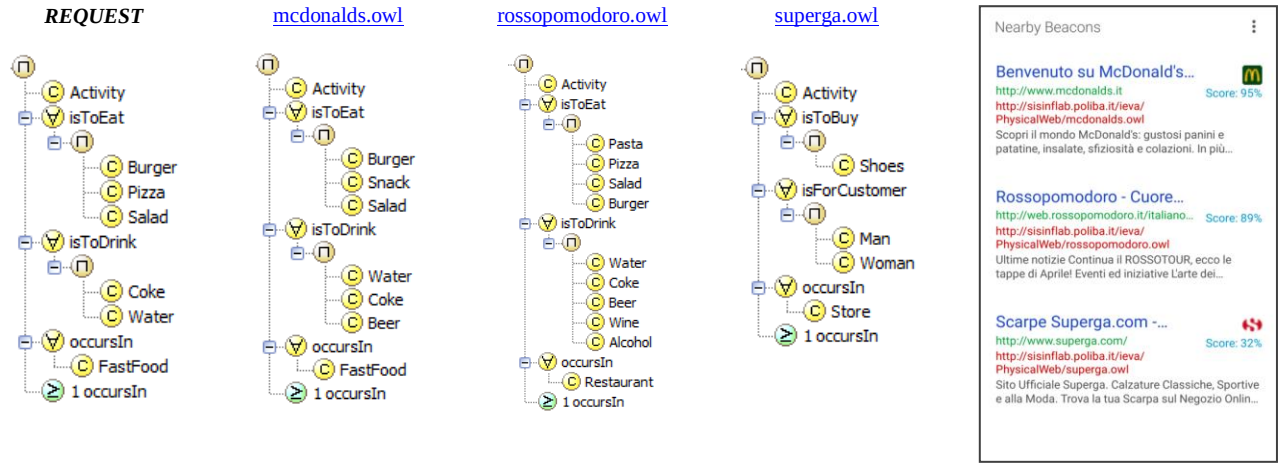


Figura 19 Risultati Scenario 2

I risultati ottenuti (Figura 19) mostrano come la risorsa McDonald's sia quella che maggiormente soddisfa la richiesta. Questo scenario è però emblematico dell'importanza dell'utilizzo della semantica nella definizione di metadati, in quanto abbiamo che:

- McDonald's non vende pizze, ma è un fastfood;
- Rosso Pomodoro soddisfa tutte le richieste di cibo e bevande, ma è un ristorante.

In un tagging privo di semantica, che non preveda disgiunzioni e annidamenti, queste due risorse avrebbero lo stesso score! Infatti, è necessario soffermarsi sulla specifica della proprietà *occursIn*: nell'ontologia utilizzata, i concetti sottoclassi di *Business*, sono tra loro **disgiunti**; per questo motivo, il calcolo dello score di RossoPomodoro viene effettuato attraverso l'utilizzo preventivo della **contrazione** (cfr. Paragrafo 2.4), che va a penalizzare maggiormente questa risorsa rispetto al caso in cui semplicemente un concetto non compare nella risorsa stessa. Quindi, un utente che vuole, ad esempio, un pranzo rapido e basso prezzo che preveda un hamburger, vedrà più in alto il beacon di un fast food, piuttosto di quello di un ristorante dove lo servono al piatto e probabilmente ad un maggiore costo.

3.4 Scenario 3

Nel terzo e ultimo scenario, l'utente richiede una diversa selezione di cibi e bevande, all'interno di un ristorante, definite dalla request mostrata di seguito:

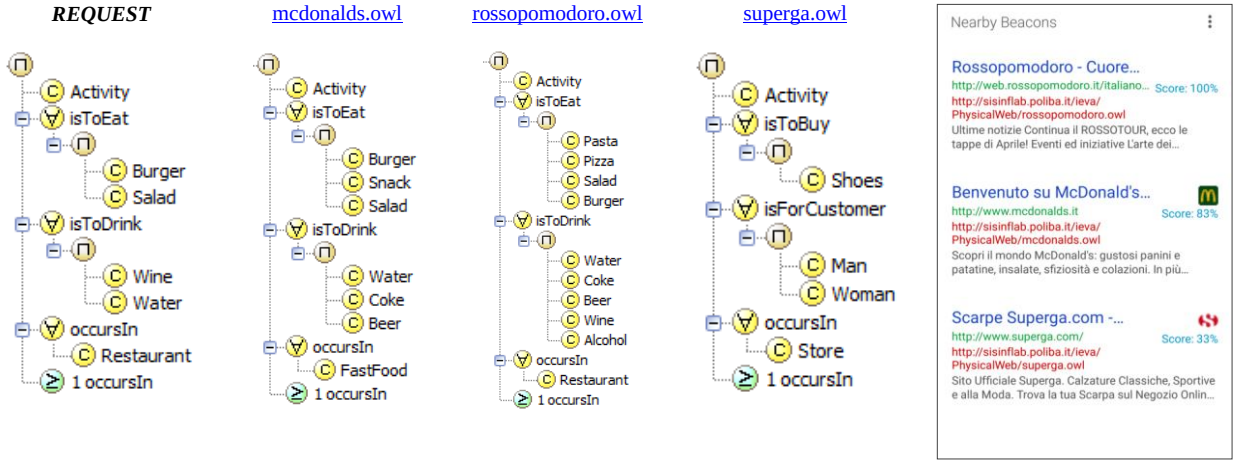


Figura 20 Risultati Scenario 3

I risultati (Figura 20) mostrano come la disgiunzione tra *Restaurant* e *FastFood* provochi un netto degrado nello score della risorsa McDonald's, sebbene essa non sia in grado di soddisfare soltanto il concetto *Wine*. E' evidente come, soprattutto a fronte di ontologie e scenari più complessi o articolati, **l'utilizzo della semantica assume un ruolo fondamentale nel soddisfacimento delle richieste di un utente all'interno di scenari variegati e modellabili in maniera approfondita.**

Sviluppi futuri

Come successivo step, l'idea di base sarebbe quella di specificare, in ciascun individuo esposto dai beacon, un'AnnotationProperty contenente l'URL di un'ontologia che descrive più in dettaglio il dominio di interesse del negozio di riferimento. **L'utente potrà dunque entrare nel negozio ed effettuare il discovery,** ad esempio**, degli scaffali presenti intorno a lui, ed ordinarli secondo una richiesta specifica fatta sull'ontologia "di secondo livello".**

Un esempio potrebbe essere il seguente:

- L'utente chiede un negozio di giocattoli sull'ontologia vista in precedenza e gli viene restituito in cima alla lista un ToyCenter vicino a lui.
- Il beacon esposto contiene un'AnnotationProperty che punta a *http://sisinflab.poliba.it/ieva/PhysicalWeb/toy.owl*, che descrive il dominio "negozio di giocattoli" più nel dettaglio.
- A questo punto entra nel negozio e fa il discovery degli scaffali, chiedendo, ad esempio, un giocattolo di tipo "Peluche", descritto nell'ontologia toy.owl, ottenendo in cima alla lista il beacon con lo scaffale dei peluche.
- All'interno dell'annotazione esposta, sarà presente un URL che punta alle offerte del mese per quello scaffale. (In caso di beacon più specifico, che si riferisce ad un oggetto specifico, l'URL potrebbe puntare ad una pagina informativa o di acquisto online di tale oggetto).

Un' alternativa sarebbe quella di permettere che ci sia un'AnnotationProperty associata ad una particolare classe dell'individuo, in modo che ogni negozio possa segnalare un'ontologia di secondo livello per la classe "Toy", ed un'altra diversa per la classe "Apparel", **nel caso in cui venda prodotti di tipo sostanzialmente diverso.**

In aggiunta, si potrebbe anche pensare ad un paradigma di utilizzo del framework leggermente diverso rispetto a quello proposto in questo lavoro. Il discovery potrebbe essere fatto sulla base di una richiesta che rappresenti **un profilo generale dell'utente**, ossia che contenga le preferenze relative a cibo, abbigliamento, divertimenti, e così via. Una volta raccolti in beacon in una certa area, si potrebbe cercare di **categorizzarli in base al tipo di attività svolta** (ad esempio utilizzando le sottoclassi del concetto *Activity* dell'ontologia utilizzata in questo lavoro), restituendo dunque una lista ordinata di beacon per ciascuna categoria. L'utente potrà dunque in qualsiasi momento visualizzare i beacon che potrebbero maggiormente fittare le sue preferenze, a seconda del tipo di servizio di cui ha bisogno in quel momento.

Ovviamente in questo scenario si pongono alcuni problemi relativi all'utilizzo di ontologie multiple, ed è inoltre necessario valutare nel dettaglio cosa accade durante il calcolo dello score per ciascuna categoria. Alternativamente, **sarebbe già possibile ottenere un approccio simile combinando più richieste**, una per ciascuna categoria, andando a calcolare lo score solo verso i beacon che corrispondono alla categoria considerata.

Bibliografia

[1] M.Ruta, F.Scioscia, E. Di Sciascio, Enabling *the Semantic Web of Things: framework and architecture*, 2012 IEEE Sixth International Conference on Semantic Computing

[2] M. Ruta, F. Scioscia, A. Pinto, E. Di Sciascio, F. Gramegna, S. Ieva, G. Loseto, *Resource annotation, dissemination and discovery in the Semantic Web of Things: a CoAP-based framework*, 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing

[3] M. Ruta, F. Scioscia, G. Loseto, F. Gramegna, S. Ieva, E. Di Sciascio, *Mini-ME 2.0: powering the Semantic Web of Things*, 3rd OWL Reasoner Evaluation Workshop (ORE 2014), Volume 1207, page 8--15 - jul 2014

[4] G. Loseto, M. Ruta, F. Scioscia, E. Di Sciascio, M. Mongiello, *Mining the user profile from a smartphone: a multimodal agent framework,* 14th Workshop From Objects to Agents (WOA 2013), Volume 1099, page 47-53 - dec 2013

[5] M. Ruta, E. Di Sciascio, F. Scioscia, *Concept Abduction and Contraction in Semantic-based P2P Environments*, Web Intelligence and Agent Systems, Volume 9, Number 3, page 179--207 - 2011