POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Polo Regionale di Como


**Corso di Laurea Specialistica in Ingegneria dell'Informazione**

# Geoservizi web OGC per il monitoraggio ambientale

l'esempio dei dati ARPA della città di Milano per l'analisi della qualità dell'aria (anni 1999-2008).


Tesi di Laurea di: **Andrea Marelli**

Matricola **673613**

Relatore: prof.ssa **Maria Antonia BROVELLI**

Correlatore: dott. **Marco NEGRETTI**



**Anno Accademico 2009/2010**

# Indice Generale

# Indice delle Figure

# Indice delle Tabelle

# Abstract

Il lavoro presentato nell'elaborato di tesi si è posto come obiettivo quello di implementare su un caso di studio reale e di interesse generale alcuni geoservizi internet al fine di verificarne e presentarne caratteristiche e potenzialità.

I servizi implementati, tutti standard proposti *dall'OGC* (*Open Geospatial Consortium*), sono il *Sensor Observation Service*, che si propone come metodo per la pubblicazione e l'accesso a informazioni e misurazioni effettuate da sensori, e il *Web Processing Service* che offre la possibilità di condividere l'accesso a funzioni, calcoli e modelli computazionali che operano su dati georeferenziati.

Nello specifico è stato realizzato un sistema che implementa i suddetti geoservizi e che permette la pubblicazione dei dati raccolti dall'*Agenzia Regionale per la Protezione dell'Ambiente della Lombardia* (*ARPA Lombardia*), nell'ambito del monitoraggio della qualità dell'aria, grazie ad una costellazione di sensori sparsi per il territorio lombardo in grado di misurare le concentrazioni nell'aria dei principali agenti atmosferici inquinanti.

# Introduzione

Nell'ultimo decennio, una delle tecnologie più innovative sviluppatesi in campo informatico, è stata sicuramente quella dei ***Web Services***. La crescente esigenza di razionalizzazione dei costi tramite un miglior sfruttamento dei dati disponili ha indubbiamente giocato uno ruolo di primaria importanza nel successo che questo tipo di applicazioni hanno avuto, e sicuramente i web services hanno dato un notevole impulso allo sviluppo ed alla crescita in numerosi campi di applicazioni.

La chiave del successo di questo tipo di applicazioni risiede proprio nel concetto stesso di servizio web e nelle potenzialità che la sua natura offre: una estrema standardizzazione, una completa indipendenza dalla piattaforma e dall'implementazione del servizio e una elevata semplicità di utilizzo. I servizi web sono infatti sviluppati in maniera tale da poter essere facilmente interconnessi ad altri servizi, favorendo la nascita e la creazione di una rete di servizi cioè di sistemi più grandi e complessi in grado di offrire una integrazione ed una possibilità di correlare dati, informazioni e funzioni, che i diversi componenti, presi separatamente, non sarebbero in grado di offrire.

Nel campo dei geoservizi, cioè tutti quei servizi che in qualche modo hanno a che fare con un'informazione di tipo geografica, l'*Open Geospatial Consortium (OGC)* ha sicuramente giocato un ruolo determinante, imponendosi come promotore nello sviluppo di standard per la creazione di servizi geo-localizzati. Lo sviluppo degli *OGC Web*

*Services (OWS)* **-** un framework di servizi web-based orientati all'accesso, all'integrazione, all'analisi, allo sfruttamento e alla visualizzazione di dati geografici, di informazioni derivate da sensori e allo sviluppo di funzioni di geo-processing – ha infatti permesso, e permette tuttora la diffusione di applicazioni sviluppate attraverso una catena di servizi dinamicamente connessi ed interoperabili.

Tra le numerose specifiche pubblicate dall'OGC, quello su cui è ricaduta l'attenzione di questa tesi è il ***Sensor Observation Service*** (***SOS***), un servizio incluso nel framework *Sensor Web Enablement* (SWE), che si propone come metodo per la pubblicazione e l'accesso alle informazioni e alle misurazioni effettuate da sensori.

Nonostante l'elevato progresso tecnologico nel campo delle comunicazioni e delle tecnologie di osservazione di fenomeni, e nonostante i concetti di *Sensor Web* e *Sensor Network* – i quali fanno riferimento ad una rete connessa al web di sensori che misurano qualunque tipo di fenomeno spazialmente distribuito ed accessibile tramite protocolli e API standard – siano stati concepiti da almeno un decennio, soltanto nell'ultimo quinquennio si è incominciato ad assistere alle prime implementazioni.

A fronte di una considerevole quantità di dati misurati in ogni istante ed in ogni parte del nostro pianeta, ad oggi non si è ancora in grado di sfruttare appieno l'enorme potenziale contenuto in queste informazioni. Basti pensare a tutte le possibili applicazioni, ad esempio in campo scientifico, ambientale, energetico, nella protezione civile o nella sicurezza, che queste informazioni raccolte continuamente sul territorio posso offrire.

Ciò che lo SWE si prefigge è la creazione di una connessione tra questi due aspetti permettendo una maggiore condivisione di tutti quei dati che consentono di studiare la realtà fornendo le capacità e gli

strumenti per una maggiore interoperabilità e correlazione tra diversi fenomeni.

Altro standard dalle enormi potenzialità e su cui si è incentrato il lavoro di questa tesi è il ***Web Processing Service (WPS)***, che permette di condividere funzioni, calcoli e modelli computazionali pre-programmati che operano con dati georeferenziati. Si pensi ad esempio, alle possibilità offerte da un servizio che solleva il client da onerosi processi, soprattutto in un periodo in cui i dispositivi mobili, indubbiamente pratici ma dalle scarse prestazioni computazionali, si stanno diffondendo con ritmi impressionanti, oppure ad un servizio che esonera un sviluppatore dal dover implementare complessi algoritmi e a procurarsi enormi moli di dati per servire tali algoritmi.

In definitiva questi servizi sono in grado di offrire un potenziale enorme, offrendo già oggi degli ottimi strumenti per implementare tali potenzialità. L'obiettivo di questo elaborato è quello di esplorare i possibili campi di applicazione di questi strumenti al fine di permettere un miglior sfruttamento di dati che oggi sono ampiamente disponibili.

I dati sugli agenti atmosferici inquinanti, raccolti quotidianamente da ormai diversi anni da una rete di sensori sparsi per il territorio lombardo dalla ***Agenzia Regionale per la Protezione dell'Ambiente della Lombardia*** (*ARPA Lombardia*) sono perciò risultati un'ottima opportunità , soprattutto in un periodo in cui una crescente sensibilità ambientale ha richiamato l'attenzione sull'inarrestabile processo di urbanizzazione ed il conseguente aumento di emissione, nocive per la salute dei cittadini.

Nel capitolo 1 verrà affrontata una breve introduzione ai servizi web geografici e una veloce panoramica dei geoservizi più diffusi ed il loro funzionamento.

Nel capitolo 2 si approfondiranno i servizi *Sensor Observation Service* (*SOS*) e *Web Processing Service* (*WPS*), oggetto della presente tesi, ed in particolare si analizzeranno in dettaglio le loro principali caratteristiche, la loro struttura ed il loro funzionamento.

Nel capitolo 3 verrà fornita una descrizione dei dati utilizzati per la realizzazione e l'implementazione dei geoservizi, analizzando la loro provenienza, le loro caratteristiche e la loro struttura.

Nei capitoli 5 e 6 si introdurranno ed analizzeranno le principali tecnologie utilizzate per implementare i geoservizi: *istSOS* per il *Sensor Observation Service* e *pyWPS* per il *Web Processing Service.* Si affronterà poi la descrizione del processo di implementazione sia dei geoservizi, che dell'interfaccia web realizzata che permette ad un qualsiasi utente di poter accedere e sfruttare le potenzialità di tali servizi.

Nel capitolo 7 si effettuerà una breve analisi delle potenziali prestazioni offerte dai geoservizi implementati.

# 1 Servizi Geografici

## *1.1 I Web Services*

I Web Services, sono un tipo di applicazioni web che cooperano fra loro attraverso lo scambio di messaggi, sono indipendenti dalla piattaforma sulla quale si trovano e dalle tecnologie con cui sono stati implementati. I Web Services sono " [...] *un nuovo tipo di applicazioni Web. Essi sono applicazioni autonome e auto-descrittive, che possono essere pubblicate, collocate e invocate attraverso il web. I servizi web svolgono funzioni, le quali possono essere qualunque cosa, dalle più semplici richieste ai più complessi processi aziendali e scientifici." In breve i Web Services, promettono di fornire un meccanismo standard per l'integrazione e l'interoperabilità tra sistemi disparati, e la chiave della loro utilità è la loro standardizzazione. [...]* (IBM Developer Works).

I servizi web sono perciò un tipo di applicazioni che presentano una serie di caratteristiche standard. Idealmente i web services presentano le seguenti proprietà:

- sono ricercabili e pubblici;
- sono auto-contenuti, cioè ogni servizio è ben definito, completo ed indipendente;
- possiedono un'interfaccia indipendente dall'implementazione;

- sono a “grana grossa”, cioè mettono a disposizione poche funzionalità ma complete, in modo da ridurre la complessità.
- permettono la l'integrazione con altri servizi.

Le suddette proprietà hanno reso i servizi web ideali soggetti per lo sviluppo di un nuovo tipo di paradigma architetturale: la *Service-Oriented Architecture* (*SOA*). Una SOA è un modello architetturale che identifica la creazione di sistemi distribuiti che focalizza l'attenzione sul concetto di *servizio*. Un sistema realizzato seguendo tale principio, è perciò costituito da applicazioni, denominate *servizi*, ben definite e completamente indipendenti fisicamente e funzionalmente una dall'altra: ognuna di esse mette a disposizione un particolare dato o funzionalità e, potenzialmente, può sfruttare quelle messe a disposizione dagli altri, realizzando così applicazioni di maggiore complessità.

La SOA comunque rappresenta soltanto un'astrazione, una architettura concettuale che non fa riferimento ad alcuna tecnologia specifica: essa si prefigge soltanto di definire le condizioni che i vari componenti, i servizi, devono rispettare.

## *1.2 Il Funzionamento dei Web Service*

La caratteristica fondamentale di un servizio web è la sua elevata standardizzazione: tramite l'utilizzo di diversi linguaggi standard, come ad esempio l'XML (Extensible Markup Language), possiede infatti, una interfaccia per la comunicazione e lo scambio di dati ben definita e comprensibile da qualsiasi client che supporta il protocollo.

Un servizio è tipicamente un'applicazione identificata da un *Uniform*

*Resource Identifier* (*URI*) la cui interfaccia è pubblica e descritta in documenti XML. La comunicazione avviene per mezzo di protocolli Internet, generalmente HTTP, la quale prevede l'interazione tra due diversi attori, il *service provider*, che identifica chi fornisce il servizio ed il *service consumer*, cioè colui che fa richiesta di tale servizio e prevede lo scambio di due messaggi, una *service request* ed una *service response* i quali devono essere definiti in modo tale da essere comprensibili alla rispettiva controparte.

Non sempre, i ruoli di provider e consumer sono ben definiti: può succedere infatti che un particolare service provider possa assumere anche il ruolo di service consumer, nel momento in cui per svolgere il proprio processo, necessita di un servizio terzo, messo a disposizione da altri.

## *1.3 L'Open Geospatial Consortium*

L'*Open Geospatial Consortium (OGC,* http://www.opengeospatial.org*)* è una organizzazione internazionale non-profit, basata sul consenso volontario, che si occupa di incoraggiare lo sviluppo e l'implementazione di standard per lo scambio ed il processing di dati e servizi geo-spaziali. Si tratta di un consorzio di oltre 400 membri tra cui società private, agenzie governative e università di tutto il mondo, che partecipano allo sviluppo di standard pubblici e gratuiti che permettono di realizzare soluzioni che “georeferenziano il Web” creando complessi sistemi di dati o servizi spaziali accessibili ed utilizzabili da qualsiasi tipo di applicazione.

La maggior parte dei prodotti sviluppati dall'OGC, principalmente documenti tecnici che descrivono in dettaglio interfacce e specifiche di

encoding, vengono indicati con il termine di *OpenGIS® Standard and Specifications*. Tali documenti sono poi utilizzati dagli sviluppatori per implementare il supporto a tali interfacce e codifiche all'interno dei loro prodotti o servizi.

Come già accennato nell'introduzione, l'OGC nell'ultimo decennio è riuscito a imporsi come promotore per lo sviluppo di standard per i servizi geografici, sviluppando e continuando tutt'oggi a sviluppare un numero sempre crescente di specifiche. Inoltre, grazie anche alla strette relazioni con l'*ISO/TC21* – la commissione tecnica formata all'interno dell'*International Organization for Standardization (ISO)* che si occupa di standard nell'area dell'informazioni geografiche digitali e di geomatica – alcune delle specifiche realizzate dall'OGC sono anche standard ISO.

## 1.4 Panoramica dei principali Standard OGC

Tra le numerose specifiche sviluppate, alcune risultano essere molto apprezzate e perciò largamente utilizzate. Nei prossimi paragrafi si farà una panoramica descrivendo brevemente alcune tra le specifiche più conosciute.

### 1.4.1 Web Map Service (WMS)

Lo standard WMS fornisce una semplice interfaccia HTTP in grado di fornire all'utente una carta georeferenziata creata con dati provenienti da uno o più database geospaziali distribuiti. La `response` di questo servizio è una o più immagini che possono essere visualizzate in un browser oppure in un applicativo GIS eventualmente combinate con

altri layer definiti localmente oppure provenienti da ulteriori server WMS. Le immagini trasferite possono essere di diverso formato (PNG, GIF, Jpeg, ...), a seconda delle richieste del client.

Le `request` che è possibile effettuare ad un server sono:

- `GetCapabilities`: ritorna i metadati del servizio, cioè informazioni circa il servizio offerto ed i layer disponibili
- `GetMap`: ritorna la carta in base ai parametri richiesti (layers, boundig-box, formato immagine, trasparenza, ...)
- `GetFeatureInfo`: permette di interrogare un layer (per quelli in cui tale funzione è disponibile) ottenendo ulteriore informazioni.

Attualmente la specifica si trova alla versione v.1.3.0.

### 1.4.2 Web Feature Service (WFS)

Il servizio WFS funziona in maniera del tutto simile al WMS ma, anziché restituire un dato sotto forma di immagine già renderizzata, il dato viene restituito codificato nel formato *Geography Markup Language* (*GML*). Lo standard GML è anch'esso sviluppato dall'OGC e consiste in una specifica che permette la rappresentazione di dati geografici, dette *features,* attraverso una sintassi basata su XML: ogni *feature* è definita da una tupla [*nome,tipo,valore*] in cui il *valore* può essere una rappresentazione di un oggetto geometrico.

Le possibili `request` che possono essere effettuate sono `GetCapabilities`, `GetFeature`, `DescribeFeatureType`, `LockFeature`, `Transaction`, and `GetFeatureWithLock`. Di queste quelle principali sono:

- `GetCapabilities`: fornisce un documento XML che descrive il servizio e quali *features* sono disponibili.

- `GetFeature`: permette di richiedere ed ottenere una *feature* in base ad alcuni parametri quali bounding-box, srs (proiezione), outputformat, etc...
- `DescribeFeatureType`: permette di ottenere una descrizione dettagliata delle varie *features* offerte dal servizio.

La versione attualmente rilasciata del WFS è la v.1.1.0

### 1.4.3 Web Coverage Service (WCS)

Il servizio WCS fornisce invece un'interfaccia standard che permette l'accesso e la condivisione di un particolare tipo di dato geo-spaziale definito *coverage*. Con il termine *coverage* ci si riferisce tipicamente a dati come immagini satellitari, fotografie aeree digitali, dati di elevazione cioè dati digitali georeferenziati che rappresentano fenomeni variabili nello spazio. Ciò significa che non vengono trasferite immagini PNG o GIF finite come succede per il WMS, ma vere e proprie griglie di dati georeferenziate, la quali possono poi essere utilizzate ad esempio, per un rendering client-side oppure come input per elaborazioni di modelli scientifici.

Il WCS fornisce tre diverse operazioni:

- `GetCapabilities`: restituisce una descrizione del servizio e dei diversi *coverage* che l'utente potrebbe richiedere.
- `DescribeCoverage`: permette di ottenere una descrizione più dettagliata di uno o più *coverage*.
- `GetCoverage`: è l'operazione che permette di ottenere il dato vero e proprio in base a parametri definiti al client. Il funzionamento è quindi molto simile alle richieste `GetMap` nel WMS ed `GetFeature` nel WFS.

La versione attualmente in uso della specifica è la v.1.1.2.

### 1.4.4 Web Processing Service (WPS)

Il servizio WPS (versione 1.0.0) si occupa di definire la sintassi su come un qualsiasi client possa richiedere l'esecuzione di un processo compresa la gestione degli input e degli output di tale processo. Una descrizione dettagliata verrà comunque affrontata nel capitolo successivo.

## *1.5 Il Sensor Web Enablement (SWE)*

A differenza degli standard precedentemente descritti, il *Sensor Web Enablement* non è un singolo standard ma una suite di differenti specifiche. Essi sono stati definiti con lo scopo di fornire tutti gli strumenti necessari per sfruttare le capacità e le potenzialità di una rete di sensori, cioè di dispositivi spazialmente distribuiti e connessi alla rete internet (Sensor Web, figura 1) in grado di monitorare diversi tipi di fenomeni come temperatura, pressione, agenti inquinanti, immagini satellitari, etc.

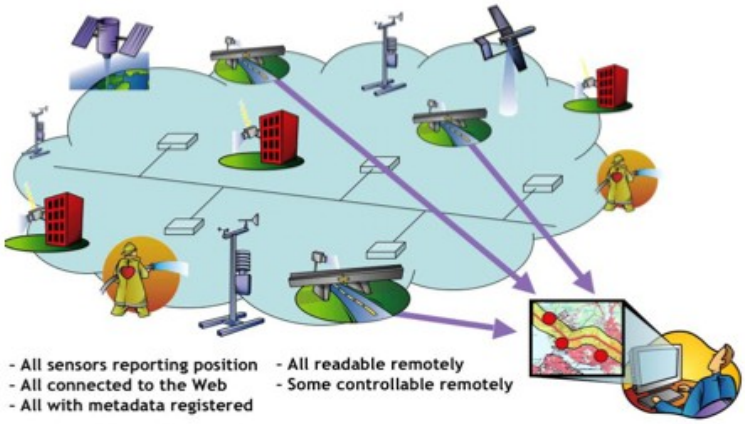


***Figura 1**: Il concetto di Sensor Web (immagine tratta da OGC White Paper - OGC® Sensor Web Enablement: Overview And High Level Architecture.)*

I protocolli ed i servizi offerti dall'architettura SWE permettono di implementare reti di sensori eterogenei, interoperabili, scalabili ed orientate ai servizi in grado di rilevare nuovi sensori, scambiare ed elaborare osservazioni ed affidare compiti. Le principali funzionalità che l'OGC ha mirato a sviluppare sono perciò:

- la scoperta e la rilevazione di nuovi sensori, osservazioni che vanno incontro alle necessità dell'applicazione o dell'utente;
- la determinazione delle capacità di un sensore e della qualità di una misurazione;
- l'accesso ai dati di un sensore che permettano la geo-localizzazione dell'osservazione;
- il recupero di dati in tempo reale o di serie temporali di dati attraverso una codifica standardizzata;
- il controllo di sensori mediante l'affidamento di compiti (task) in grado di fornire misurazioni;
- la sottoscrizione e la pubblicazione di allarmi che dovranno essere

emessi dai sensori in particolari circostanze;

Tutte le sette specifiche che fanno parte dello standard SWE utilizzano l'XML per pubblicare le descrizioni e le capacità dei sensori e si approcciano al concetto di sensore e dato attraverso un paradigma *object oriented* il quale fornisce il modo di generare strutture di dati, prodotti dai sensori, comprensibili che facilitano l'interpretazione di dati in archivi distribuiti.

Le specifiche incluse nello standard SWE sono qui di seguito elencate e brevemente descritte:

- ***Observation & Measurements Schema (O&M)***: specifica un modello ed una codifica XML per la descrizione e la rappresentazione di misurazioni, di osservazioni ed di metadati dei sensori.
- ***Sensor Model Language (SensorML)***: descrive un modello ed un codifica che permette la rilevazione di sensori, la loro descrizione e quella delle relative osservazioni. SensorML fornisce anche la possibilità di dettagliare i componenti dei sensori in modo tale da descrivere anche il processo con cui le misurazioni delle osservazioni vengono effettuate.
- ***Transducer Markup Language (TML)***: fornisce un metodo e una codifica a supporto della cattura, dello scambio e dell'archiviazione di dati da e verso i sistemi di sensori. Esso definisce un set di modelli per descrivere le caratteristiche della risposta hardware di un trasduttore e un metodo per trasportare efficientemente i dati e prepararli all'associazione con i dati temporali e spaziali.
- ***Sensor Observation Service (SOS)***: fornisce un metodo per la pubblicazione e l'accesso alle informazioni e alle misurazioni effettuate da sensori. Essendo comunque tema di particolare rilevanza per il presente elaborato, una descrizione più dettagliata

verrà fornita nel corso del prossimo capitolo.

- ***Sensor Planning Service (SPS)***: fornisce un servizio che permette a dei client di determinare la possibilità e di pianificare la richiesta di una particolare collezione di dati per uno o più sensori
- ***Sensor Alert Service (SAS)***: fornisce una interfaccia per richiedere informazioni circa la natura degli allarmi offerti, i protocolli utilizzati e le opzioni per “iscriversi” ad uno specifico tipo di allarme.
- ***Web Notification Services (WNS)***: specifica il servizio attraverso cui un client può condurre un scambio di messaggi asincrono con uno più servizi

# 2 Servizi SOS e WPS

## 2.1 Il Sensor Observation Service (SOS)

Come già accennato nel capitolo precedente, il *Sensor Observation Service* (*SOS*) è un servizio che fornisce una interfaccia per ricercare, gestire e ottenere dati sia in tempo reale che archiviati, prodotti da sensori di diverso tipo, sia mobili che stazionari, sia remoti che in-situ. I dati che è possibile ottenere dai sensori possono essere sia misurazioni che descrizioni di sensori: le misurazioni, dette *osservazioni,* sono codificate per mezzo del *O&M*, mentre le informazioni relative ai sensori sono ritornate come SensorML oppure TML. Un servizio SOS, quindi, utilizzato in stretta relazione con gli standard OGC appartenenti allo SWE, costituisce un ambiente di capacità e strumenti per connettere ed interrogare sensori, singoli o appartenenti ad una rete.

L'obbiettivo primario del SOS comunque è quello di permettere l'accesso alle *osservazioni.* Un'*osservazione (observation)* è un'azione associata ad preciso istante di tempo o ad un periodo, il cui valore è assegnato ad un fenomeno *(phenomenon).* Le proprietà fondamentali di un'osservazione sono la ***featureOfInterest*** (caratteristica di interesse), la ***observedProperty*** (proprietà osservata), la ***procedure*** (procedure) e il ***result***.

La *featureOfInterest* è una caratteristica di qualsiasi tipo (ISO 19109, ISO 19101) obiettivo dell'osservazione. La *observedProperty* identifica o descrive il fenomeno per cui l'osservazione fornisce una stima del suo valore. Deve essere una proprietà associata con il tipo della caratteristica di interesse. La *procedure* è la descrizione del processo usato per generare il risultato e varia a seconda del tipo di proprietà osservata. Spesso è uno strumento, ad esempio un sensore ma può anche essere un osservatore umano, un algoritmo, un calcolo oppure una simulazione. L'elemento *result* contiene il valore generato dalla procedura il cui type (tipo) può essere di qualsiasi genere, ma consistente con la proprietà osservata.

Le operazioni che possono essere eseguite con il Sensor Observation Service, sono suddivise in quattro differenti gruppi, detti *profili*. Il *Core Profile* è composto da tre operazioni di base che rappresentano il requisito minimo per poter considerare un servizio conforme allo standard SOS; il *Transactional Profile* è invece opzionale ed include tutte le funzioni necessarie per registrare nuovi sensori ed inserire nuove osservazioni nel servizio; l'*Enhanced Profile*, anch'esso opzionale, fornisce una serie di operazioni aggiuntive per l'interrogazione dei sensori e delle osservazioni. L'*EntireProfile* caratterizza invece quei servizi che implementano tutte le operazioni definite nei profili descritti in precedenza. Nei prossimi paragrafi verranno descritte dettagliatamente tutte le operazioni fornite da ciascun profilo.

### 2.1.1 Core Profile

Come accennato precedentemente, il *Core Profile* include le tre operazioni obbligatorie che ciascun servizio SOS dovrebbe implementare e che forniscono le funzionalità di base per ottenere

informazioni di sensori.

La `GetCapabilities` permette ai client di ottenere come *response*, un documento XML contenente i metadati che descrivono la specifica istanza del servizio.

Tale documento include perciò:

- una sezione *ServiceIdentification* ed una *ServiceProvider* che contengono informazioni di base circa il servizio stesso ed il suo provider,
- una sezione *OperationsMetadata* contenente l'elenco delle operazione fornite dal servizio, con i relativi parametri supportati
- una sezione *FilterCapabilities* utilizzata per elencare quali tipi di parametri di interrogazione sono supportati dal servizio
- una sezione *Contents* che descrive le caratteristiche di ciascuna Observation Offering con le relative *procedure, observedProperty,* e *featureOfInterest*.

La `DescribeSensor` permette di ottenere metadati dettagliati circa i vari sensori, codificati con gli standard SensorML oppure TML. Tali documenti possono includere anche una lista dei fenomeni osservati dal sensore.

La `GetObservation` è invece utilizzata per interrogare i sensori ed ottenere dati di misurazioni nel formato O&M. Questa operazione prevede l'utilizzo di alcuni parametri che permettono di affinare la query ed ottenere dati di misurazioni più in linea con le reali necessità. I parametri più rilevanti in base ai quali si possono effettuare *request* di osservazioni sono: eventTime, cioè l'istante od il periodo su cui effettuare la query, la *procedure*, *observedProperty* e la *featureOfInterest*.

### 2.1.2 Transactional Profile

Il *Transactional Profile* fornisce all'utente due operazioni che permettono una comunicazione finalizzata alla registrazione di nuovi dati. A differenza del profilo precedentemente descritto, il t*ransactional profile* è opzionale e, perciò, non necessario al corretto sviluppo di un server SOS.

La `RegisterSensor` permette al client di registrare un nuovo sensore all'interno di un servizio SOS. Ovviamente ciò implica che nella *request* inviata dal client sia inserita la descrizione del sensore in questione: tale descrizione consiste in un documento SensorML oppure TML. Nel caso in cui la registrazione del nuovo sensore andasse a buon fine, le *response* restituita dal servizio fornirebbe l'identificatore assegnato al sensore *(AssignedSensorId).*

La `InsertObservation` fornisce invece al client una funzione che permette di inserire una nuova osservazione per un sensore precedentemente registrato. La *request* deve contenere un documento XML che segue le specifiche O&M al cui interno, oltre all'osservazione stessa, deve essere riportato l'identificatore del sensore, ottenuto dalla *RegisterSensor* in modo da poterla associare correttamente. La *response* restituita dal server SOS riporta l'identificatore assegnato alla osservazione.

### 2.1.3 Enhanced Profile

Anche per l'*Enhanced Profile,* come per il *transactional*, la sua implementazione non risulta essere obbligatoria. Fornisce all'utente una serie funzionalità aggiuntive che permettono di ottenere informazioni più specifiche riguardo ai diversi elementi che compongono il sistema: in ogni caso le informazioni che possono

essere ottenute con l'*Enhanced Profile* possono comunque essere ottenute per mezzo del *Core Profile*.

La `GetObservationByID`, permette di ottenere una particolare osservazione, la `GetResult` fornisce gli strumenti per ottenere dati da un determinato set si sensori senza dover ritrasmettere e ricevere grosse quantità di dati che già sono state ottenute. Il `GetFeatureOfInterest` permette di ottenere informazioni circa un *featureOfInterest* mentre la `GetFeatureOfInterestTime` ritorna il periodo di tempo per cui sono disponibili dati di una particolare *featureOfInterest;* la `DescribeFeatureType` ritorna la descrizione GML di una *featureOfInterest,* ed infine, la `DescribeObservationType` ritorna un XML che descrive che tipo di osservazione è ritornata per un particolare fenomeno.

### 2.1.4 Stato dell'arte

Il servizio SOS, a differenze di altri standard più comuni, risulta essere poco diffuso. Le ragioni possono essere molteplici, dalla giovane età dello standard la cui pubblicazione risale all'ottobre del 2007, alla relativa complessità di implementazione del servizio e delle specifiche SWE. In ogni caso lo standard ha tutte le potenzialità per diventare un servizio di primaria importanza nell'interscambio di dati provenienti da diverse parti del mondo in maniera semplice e veloce, aprendo nuove possibilità di studio  e di applicazione dei dati raccolti dai sensori.

Esistono comunque diversi esempi di implementazione di questo standard liberamente accessibili sulla rete.

Un esempio di implementazione dello standard SOS è dato dal *NDBC* (*National Data Buoy Center*) *SOS Server* (http://sdf.ndbc.noaa.gov/) sviluppato dal *NOAA* (*National Oceanic and Atmosferic*

*Administration's*) per il *IOOS (Integrated Ocean Observing System)* progetto a cui partecipano decine di istituti ed agenzie federali, regionali e del settore privato negli Stati Uniti, con lo scopo di migliorare le capacità di collezionare, scambiare ed utilizzare informazioni sugli oceani, sulle coste e sui laghi degli Stati Uniti. Il servizio tramite migliaia di sensori in-situ sparsi per l'Oceano Atlantico e il Pacifico è in grado di fornire dati circa la temperatura, la salinità, le correnti, il livello dell'acqua, delle onde e del vento.

Un altro esempio di implementazione del servizio è l'*HydroPortal* (figura 2, http://istgeo.ist.supsi.ch/hydro/), un portale sviluppato tramite un rete di sensori pluvio-idro-geologici nel Canton Ticino dall'*Istituto Scienze della Terra (SUPSI)* e l'*Ufficio dei Corsi d'Acqua* del Canton Ticino. L'applicazione è stata sviluppata tramite il software ***istSOS***, realizzata sempre dall'Istituto Scienze della Terra, utilizzato per implementare il servizio SOS anche nel presente elaborato: nel capitolo successivo se ne darà una descrizione esauriente.


***Figura 2:*** *HydroPortal - Istituto Scienze della Terra*

Anche per quanto riguarda lo sviluppo di software che fa uso del

suddetto servizio si è riscontrata una diffusione piuttosto ridotta. Tra i vari software che implementano il servizio di server SOS, oltre all'*istSOS,* che, come accennato verrà descritto nel dettaglio nel successivo capitolo, quelli più rilevanti sono il *52°NorthSOS* ed il *degree3*.

Il framework ***52°N Sensor Observation Service*** (http://52north.org/SensorWeb/sos/index.html) è un software sviluppato da *52°North Initiative for Geospatial Open Source Software GmbH* un gruppo di ricerca e sviluppo che opera a livello internazionale e che si occupa di promuovere l'ideazione, lo sviluppo e l'applicazione di software geografico open source nel campo della ricerca, dell'istruzione e in ambito applicativo. Il software è scritto interamente in Java, rispetta le specifiche SOS v.1.0.0 ed è in fase di approvazione come "OGC compliant" (attualmente è in fase *beta*).

***Degree3*** (http://www.deegree.org/), è invece l'ultima versione del framework degree, sviluppato da lat/lon GmbH, azienda tedesca nata dal *GIS Working Group* presso il *Dipartimento di Geografia all'Università di Bonn* ed impegnata nell'implementazione di prodotti e servizi nel settore dei GIS web-based. Il framework è sviluppato in Java e distribuito sotto licenza LGPL: la versione stabile attualmente rilasciata è la 2.3 ed implementa diversi standard OGC tra cui WMS, WPS, WFS, WCS, etc. La versione 3 del framework oltre a quelli sopraccitati implementa anche lo standard SOS (v1.0.0) e, nonostante sia ancora in piena fase di sviluppo, risulta essere l'unico software ad essere registrato presso l'OGC come "*Certified OGC Compliant*" per lo standard SOS (http://www.opengeospatial.org/resource/products).

## 2.2 Il Web Processing Service (WPS)

La specifica Web Processing Service definisce un modello di comunicazione con un servizio in grado di offrire ad un qualsiasi client l'accesso a funzioni, calcoli e modelli computazionali pre-programmati che operano con dati georeferenziati, disponibili presso il server oppure forniti come input dallo stesso client. Lo scopo del WPS è quindi quello di standardizzare il modo in cui processi geo-spaziali vengono invocati così da ridurre e facilitare notevolmente la quantità di lavoro richiesto per l'implementazione e l'adozione di tale servizio.

Come si evince, il WPS possiede una natura molto generica, e non identifica alcun processo specifico. Sarà invece compito di ciascuna istanza del servizio definire ciascun *process* descrivendone il funzionamento e i relativi input ed output. Il WPS può essere immaginato quindi, come un sorta di modello astratto di web service in cui ogni processo dovrà essere sviluppato e standardizzato per la propria l'interoperabilità.

Il servizio WPS implementa tre diverse operazioni, tutte obbligatorie ai fini dell'implementazione di un server WPS standard, le quali risultano avere molte somiglianze con altri servizi OGC. Nei prossimi paragrafi verrà descritto il loro funzionamento nel dettaglio.

### 2.2.1 GetCapabilities

Questa operazione permette ad un client di ricevere un documento contenente i metadati che descrivono le capacità del server WPS. Questo documento fornisce nome e descrizione di ogni processo disponibile sul server.

### 2.2.2 DescribeProcess

La **`DescribeProcess`** permette al client di ottenere informazioni dettagliate per ogni processo, come ad esempio i dati di input, i formati supportati, l'output restituito

### 2.2.3 Execute

L'operazione di **`Execute`** permette al client di richiedere l'esecuzione di uno specifico processo. I dati di input possono essere specificati direttamente nella *request,* possono essere disponibili sul server, oppure possono essere risorse disponibili attraverso il web. Allo stesso modo gli output del processo eseguito potranno essere ritornati in forma grezza all'interno del documento XML di *response*, oppure potranno essere messi disponibili sul web: in questo caso nella *response* sarà specificato l'URL relativo all'output.

### 2.2.4 Stato dell'arte

A differenza dello standard SOS, il WPS risulta invece essere molto più diffuso. La conferma di ciò arriva dalla sua implementazione in un numero maggiore di software ed applicativi in grado di implementare tale specifica.

Oltre al già citato ***degree3***, descritto in precedenza e al ***pyWPS***, che è stato utilizzato in questo lavoro di tesi e pertanto sarà illustrato in modo dettagliato nei capitoli successivi, vi sono altri applicativi di cui si da una breve descrizione di seguito.

La 52°North appare essere molto attiva nell'implementazione di tale specifica; oltre ad aver sviluppato ***52°n WPS*** (http://52north.org/maven/project-sites/wps/52n-wps-webapp/), un framework scritto in Java per lo sviluppo di un server WPS, ha inoltre

realizzato dei client per poter sfruttare le potenzialità del WPS all'interno di altri software tra cui plug-in per *uDig* e per *OpenJump* – due diffusi software desktop GIS open-source – e delle generiche API in java.

Oltre alle soluzioni open-source risultano esserci anche applicativi commerciali di classe enterprise, tra cui ***ERDAS APOLLO Professional*** (http://www.erdas.com/Products/ERDASProductInformation/tabid/84/currentid/3148/objectid/3148/default.aspx) una potente implementazione dello standard WPS sviluppato da *ERDAS.*

Alcuni esempi di implementazioni del servizio WPS possono essere visualizzati nel sito web dell'HS-RS (http://www.bnhelp.cz/produkty/gis-na-webu/), società che opera nel settore del telerilevamento nella Repubblica Ceca, principale sponsorizzatore del progetto pyWPS. Tra gli esempi più rilevanti si può osservare l'*Analytical Server of SEA* (http://geo.sazp.sk/), uno strumento per il calcolo delle superfici di visibilità.

# 3 Dati e modello di dati

## *3.1 ARPA Lombardia e il monitoraggio degli inquinanti atmosferici*

L'*Agenzia Regionale per la Protezione dell'Ambiente della Lombardia* (*ARPA Lombardia*) si occupa di prevenzione e protezione dell'ambiente affiancandosi ad istituzioni regionali e locali in attività tra cui la lotta all'inquinamento atmosferico ed acustico, gli interventi per la tutela delle acque superficiali e sotterranee, il monitoraggio dei campi elettromagnetici e le indagini sulla contaminazione del suolo e sui processi di bonifica. Il settore *Aria e Agenti Fisici* si occupa, in merito al processo di valutazione della qualità dell'aria, proprio al controllo delle emissioni, alla loro valutazione ed di fornire supporto nell'adozione di piani e programmi per il miglioramento della qualità dell'aria.

### 3.1.1 Gli inquinanti atmosferici monitorati

Nonostante si conoscano gli effetti provocati sulla salute da una singola sostanza non sono ancora del tutto chiare le possibili conseguenze croniche che l'esposizione per tempi prolungati a miscele di diversi tipi di agenti e con differenti concentrazioni possa avere sulla salute degli esseri viventi e sull'ambiente in generale.

L'aumento registrato negli ultimi anni di patologie alle vie respiratorie e di danni alla vegetazione, così come l'aumento del degrado della qualità dell'aria in ambienti sottoposti alla pressione antropica, hanno reso quindi indispensabile l'approfondimento delle possibili relazioni tra questi due aspetti.

Punto di partenza per tali studi è così la misurazione ed il monitoraggio costante di sostanze nocive, dette *indicatori,* con le quali è possibile stabilire il grado di inquinamento dell'aria. Di seguito verranno elencati e illustrati brevemente i principali indicatori monitorati attualmente dall'ARPA (Tabella 1).

- *PM10 e PM2,5*: con i termine PM (Particulate Matter) si definisce un mix di particelle (particolato) che si trovano in sospensione nell'aria aventi diametro inferiore rispettivamente a 10 e a 2,5 µm. La provenienza di queste sostanze, benché possa essere anche da fenomeni naturali, risulta essere in gran parte da attività antropiche quali traffico veicolare e combustione.
- *Ossidi Azoto (N0X)*: sono sostanze derivanti da processi di combustione e provengono da centrali termoelettriche, impianti di riscaldamento e principalmente da traffico veicolare. Un ruolo particolarmente rilevante è svolto dal *Biossido di Azoto (NO2)*.
- *Ozono Troposferico (O3)*: è un gas che si trova per lo più nella troposfera (tra i 10 e 50 Km di altezza). Non essendo direttamente emesso, costituisce un inquinante secondario i cui precursori sono generalmente emessi dalla combustione industriale in cui si fa uso di solventi o carburanti.
- *Monossido di Carbonio (CO)*: è un gas infiammabile e molto tossico e deriva principalmente dall'utilizzo di combustibili fossili come gas naturali, carburanti e benzina, carbone e legna.

- *Biossido di Zolfo (SO2)*: è un gas irritante derivato soprattutto dalla combustione di prodotti organici di origine fossile contenenti zolfo.
- *Benzene (C6H6)*: è il più comune, e allo stesso tempo uno dei più tossici tra gli idrocarburi aromatici; le fonti principali sono il traffico veicolare ed alcuni processi industriali.

| | | unità di misura | frequenza di campionamento |
|---|---|---|---|
| Benzene | C6H6 | µg/m3 | 1 ora |
| Monossido di Carbonio | CO | mg/m3 | 1 ora |
| Biossido di azoto | NO2 | µg/m3 | 1 ora |
| Ossido di azoto totale | NOX | ppb | 1 ora |
| Ozono | O3 | µg/m3 | 1 ora |
| Particolato 2.5 µm | PM25 | µg/m3 | 1 giorno |
| Particolato 10 µm | PM10 | µg/m3 | 1 giorno |
| Biossido di zolfo | SO2 | µg/m3 | 1 ora |

***Tabella 1:*** *Elenco delle sostanze inquinanti osservate (ARPA Lombardia)*

### 3.1.2 Le stazioni di rilevamento ed i sensori

Il monitoraggio degli agenti atmosferici inquinanti eseguito costantemente dall'ARPA Lombardia viene effettuato da una rete di più di 150 centraline sparse per tutto il territorio lombardo. Ogni stazione è costituita da un modulo in vetroresina in cui vengono alloggiati gli *analizzatori degli inquinanti atmosferici*, e tutte strumentazioni necessarie per la visualizzazione, l'elaborazione e la trasmissione dei misurazioni effettuate. Tutta la strumentazione lavora in modo totalmente automatico e continuativo durante l'arco delle 24 ore, ed è in grado di rilevare le sostanze con elevata sensibilità in grado di fornire dati attendibili anche a basse concentrazioni.

### 3.1.3 I dati utilizzati

I servizi geografici implementati con questo elaborato sono stati realizzati con lo scopo di verificarne e presentarne le caratteristiche e le potenzialità rispetto ad un case study reale ed di interesse generale. Per tale motivo i dati importati nel database ed utilizzati a titolo esemplificativo sono i dati effettivamente raccolti dall'ARPA Lombardia nell'ambito del monitoraggio dell'aria, ottenuti grazie allo strumento di download dei dati storici messo a disposizione sul sito dell'ARPA (http://ita.arpalombardia.it/ITA/qaria/doc_RichiestaDati.asp).

Di tutto l'archivio di dati disponibili si è scelto di considerare ed utilizzare soltanto i dati presi dalle centraline del comune di Milano nel decennio 1999-2008. Va sottolineato comunque, che questi limiti sono stati creati solamente per semplificare il processo di scaricamento e di gestione dei dati; ciò non vieta in alcun modo – ma soprattutto non vi è alcun impedimento di tipo tecnico od architetturale – di allargare il dominio di interesse del sistema implementato aggiungendo l'intero archivio di dati misurati dall'ARPA sull'intero territorio lombardo.

I dati ottenuti e utilizzati per l'implementazione fanno perciò riferimento soltanto alle centraline localizzate all'interno del territorio comunale di Milano e sono relativi al decennio 1999-2008 (dal 1 Gennaio 1999 al 31 Dicembre 2008). In Tabella 2 è riassunto il posizionamento delle centraline, con i relativi sensori.

| | Coordinate Geografiche (Nord,Est) | Sensori |
|---|---|---|
| Milano – P.zza Zavattari | 5035867,1511109 | C6H6, CO, NO2, NOX |
| Milano – P.zza Abbiategrasso | 5031000,1514275 | NO2, NOX |
| Milano – Parco Lambro | 5038500,1519350 | NO2, NOX, 03 |
| Milano – Pascal Città Studi | 5036210,1518432 | N02, NOX, O3, PM25, PM10, SO2 |
| Milano – Verziere | 5034463,1515297 | CO, NO2, NOX, O3, PM10 |
| Milano – via Senato | 5035258,1515462 | C6H6, CO, NO2, NOX, PM10 |
| Milano – viale Liguria | 5032293,1513161 | CO, NO2, NOX |
| Milano – viale Marche | 5038125,1514945 | CO, NO2, NOX |

***Tabella 2**: Elenco delle stazioni di rilevamento*

La proiezione utilizzata per indicare le coordinate delle stazioni è la proiezione di Gauss-Boaga introdotta nel 1940 dall'Istituto Gegrafico Militare. Il sistema di riferimento geodetico è Roma40 (Roma Monte Mario). Tale proiezione è stata classifica dal *European Petroleum Survey Group* (*EPSG*) con il i codici univoci di EPSG:3003 (ovest) e EPSG:3004 (est).

#### 3.1.3.1 Caratteristiche dei dati forniti

I dati sulle misurazioni che l'ARPA mette a disposizione sul proprio sito sono semplici elenchi di misurazioni in formato *CSV* (*Comma-Separated Values*), un formato di tipo testuale, normalmente utilizzato per l'importazione e l'esportazione di tabelle di dati. Grazie all'utilizzo di separatori e delle righe è infatti possibile organizzare i dati come se fossero in una tabella: ogni riga della tabella normalmente rappresentata da una linea di testo, che a sua volta è divisa in campi (le singole colonne) separati da un apposito carattere separatore, ciascuno dei quali rappresenta un valore. Di seguito (Figura 3) è possibile osservare un piccolo estratto di un file csv dei dati ARPA.

```
5550,1999/01/01 01:00,52
5550,1999/01/01 02:00,54
5550,1999/01/01 03:00,53
5550,1999/01/01 04:00,54
5550,1999/01/01 05:00,54
5550,1999/01/01 06:00,30
5550,1999/01/01 07:00,36
5550,1999/01/01 08:00,52
5550,1999/01/01 09:00,55
5550,1999/01/01 10:00,53
5550,1999/01/01 11:00,50
```

***Figura 3**: Esempio di dati ARPA in formato csv*

Ogni misurazione è caratterizzata quindi da un identificatore del sensore, l'istante in cui la misurazione è stata effettuata (comprensivo di data e ora) e il valore stesso della misurazione. Nel capitolo in cui si illustrerà l'implementazione dei servizi, verrà descritto il funzionamento dello script creato per estrarre in maniera del tutto automatica i dati dai file CSV e inserirli nel database tramite delle richieste di *insertObservation*.

### 3.1.4 Altri layer utilizzati

Al fine di rendere le carte visualizzate più descrittive, ed in modo da rendere il posizionamento dei risultati delle osservazioni il più esplicativo, sono stati utilizzati altri layer.

Tutti i dati sono stati prelevato dal *Geoportale della Lombardia* (http://www.cartografia.regione.lombardia.it/geoportale) sito di riferimento per i dati geografici sul territorio lombardo e sono stati prelevati tutti con la stessa proiezione utilizzata per il posizionamento delle osservazioni, cioè la Gauss-Boaga (EPSG:3003). Questo principalmente perché i dati, ed in particolar modo i raster, sono attualmente disponibili soltanto in questa proiezione.

Per meglio contestualizzare i dati delle osservazioni, sono stati così

utilizzati il raster della Carta Tecnica Regionale in scala 1:10000 come sfondo, ed i layer vettoriali della rete ferroviaria, della rete stradale, e delle aree protette.

## 3.2 Modellazione del database delle osservazioni

La creazione e modellazione del database è stata sicuramente una delle fasi più critiche dell'implementazione dei servizi geografici. Se da un lato il servizio WPS non presenta particolari requisiti in termini di struttura del database, il *Sensor Observation Service* risulta essere decisamente più esigente. Esigenze che scaturiscono indubbiamente dalla relativa complessità che caratterizza la suddetta specifica, la quale vede interagire tra loro diverse entità, e che nel software istSOS, – utilizzato per l'implementazione del servizio – si traduce nella imposizione di una specifica e ben definita struttura che il database deve possedere.

Le entità che la specifica prevede sono *offering, feature of interest*, *procedure*, *observed property* e *observation* (o *measure*) ed il primo passo è stato ovviamente quello comprendere appieno il significato di ciascuna entità e di identificare in maniera precisa i vari elementi nel dominio di applicazione a cui ci si riferisce.

In base a quanto stabilito nelle specifiche SOS, il concetto di *offering* fa riferimento alle modalità con cui il SOS raggruppa i sensori che sono in qualche modo collegati e che possono contribuire al raggiungimento di un particolare scopo: tale concetto si può in un certo senso associare a quello di “progetto” in cui vengono raccolti dati da tutti i sensori. É necessario infatti sottolineare il fatto che un sistema SOS può contenere un gran numero di sensori magari anche molto differenti tra

loro e che misurano fenomeni che non hanno niente a che vedere uno con l'altro. Nel caso in esame, il concetto di “monitoraggio della qualità dell'aria” da parte di ARPA Lombardia può perciò ben adattarsi a quello di *offering*. Sempre in base a quanto descritto dalle specifiche SOS, il concetto di *feature of interest* – già illustrato nel capitolo precedente – può essere associato a quello delle centraline di rilevamento, ognuna caratterizzata da un nome, una descrizione ed da una posizione nello spazio. In maniera del tutto simile sono stati identificati come *procedure* i singoli sensori, ognuno associato ad una specifica *feature of interest* ed in grado di rilevare una specifica *observed property*, e cioè gli agenti atmosferici inquinanti come NOX, CO, PM10, etc.. Decisamente più immediato è stato invece identificare il concetto di *observation* (*measure*). Nella figura 4 si può quindi osservare la possibile struttura del database considerando soltanto gli elementi di base appena descritti.

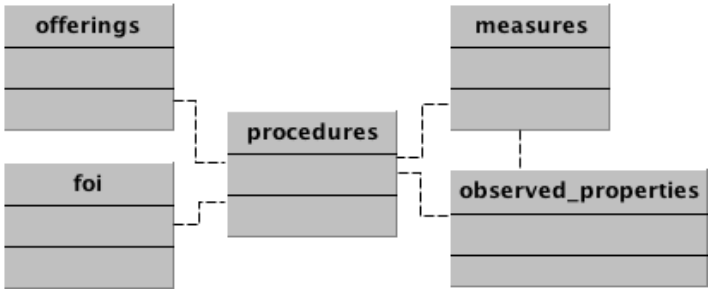


***Figura 4**: Struttura di base del database delle osservazioni*

Da questa struttura di base si passa poi, attraverso un processo di raffinazione, ad una struttura decisamente più complessa, in cui sono forniti tutti i campi necessari ad accogliere le informazioni necessarie ad una corretta implementazione e al funzionamento nel rispetto delle specifiche del servizio. L'architettura definitiva, proposta dagli sviluppatori del software istSOS risulta perciò quella rappresentata in figura 5 .

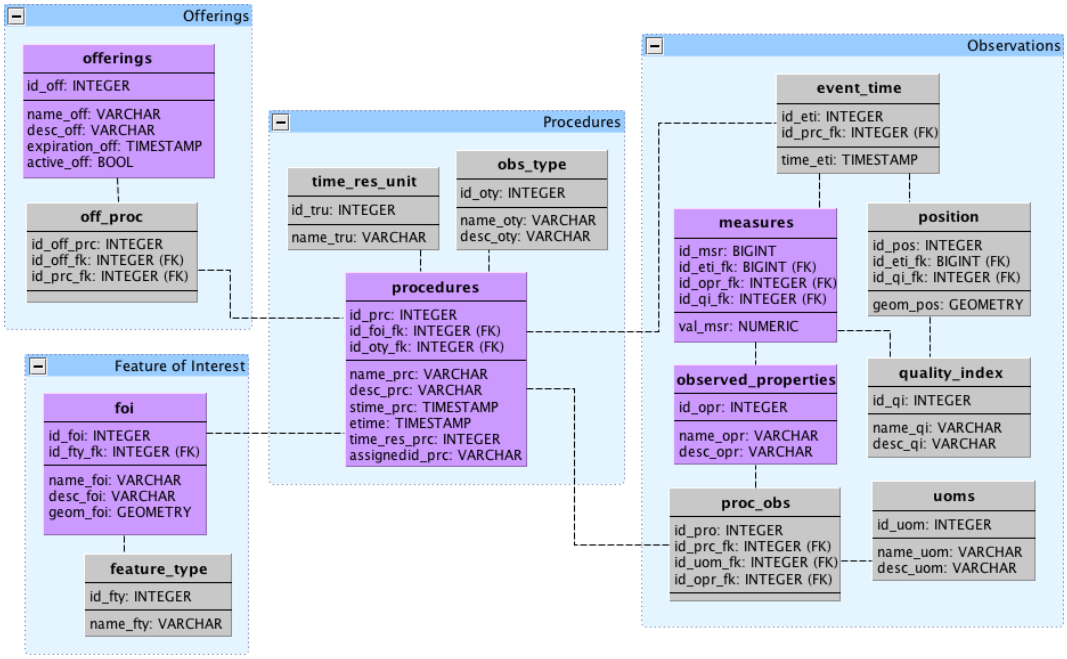


***Figura 5**: Schema UML del database delle osservazioni*

Nonostante la struttura del database per le osservazioni appena descritta sia ben organizzata è con molto probabilità simile a quella che si sarebbe ottenuto nel caso fosse stata necessaria una attenta modellazione del database ex-novo, il fatto che essa sia stata in un certo senso imposta rimane una grossa limitazione, in particolar modo se si considera il caso dell'implementazione del servizio in un sistema già operativo e con un proprio database caratterizzato da una propria struttura. D'altro canto, una struttura rigida e che impone il suo popolamento solo attraverso gli strumenti propri del servizio (*registerSensor* e *insertObservation*) assicura una maggiore aderenza alle specifiche.

# 4 Implementazione servizi geografici

## 4.1 Configurazione del server

Dato il migliore supporto e la migliore integrazione di alcuni software utilizzati per la configurazione del web server e dei geoservizi con sistemi Linux, è stato scelto come ambiente di sviluppo il sistema *Ubuntu*, una delle distribuzioni Linux più diffuse.

La maggior parte del software necessario alla corretta configurazione del server, è stato possibile installarlo direttamente attraverso pacchetti pre-compilati forniti dai repositories ufficiali di Ubuntu. Con questo metodo è stato possibile installare l'*Apache HTTP web server*, l'interprete *PHP (*con il relativo modulo per Apache) ed il *mod_python*, un modulo che integra l'interprete Python nel web server *Apache* (l'interprete Python è solitamente già installato su molte distribuzioni linux). É stato poi installato il database *PostgreSQL*, con l'estensione *PostGIS* per la gestione di dati geografici ed infine *MapServer* e *GRASS*, con tutte le librerie da cui dipendono: tra queste vi sono *GDAL* (libreria per la gestione di molteplici formati raster), *OGR* (libreria per l a gestione di dati vettoriali), *PROJ.4* (per la gestione di proiezioni cartografiche) e *numpy* (un pacchetto che fornisce a Python strumenti per il calcolo scientifico).

É stata invece necessaria la compilazione dei software istSOS e pyWPS, utilizzati per l'implementazione dei due servizi geografici, la cui configurazione ed implementazione verrà analizzata nei prossimi paragrafi.

## *4.2 istSOS*

Il software ***istSOS*** (*Istituto Scienze della Terra Sensor Observation Service, http://istgeo.ist.supsi.ch/software/istsos*) è un'implementazione dello standard OGC Sensor Observation Service sviluppato dall'*Istituto Scienze della Terra*. L'istituto si occupa di temi legati all'ambiente e fa parte del Dipartimento Ambiente Costruzioni e Design della *Scuola Universitaria Professionale della Svizzera Italiana* (*SUPSI*): i settori in cui opera riguardano l'idrologia, l'idrogeologia, la geologia, la geotecnica e la geomatica.

Lo sviluppo del software è iniziato nel 2009, con l'intento si sviluppare una implementazione la più semplice possibile del servizio SOS per la gestione, la misurazione e l'integrazione di dati idro-meteorologici raccolti nel Canton Ticino e ha portato al rilascio di una prima release candidate all'inizio del 2010 e di una seconda lo scorso maggio.

Il software è scritto interamente in ***Python*** ed il suo funzionamento si basa su di un database *PostgreSQL*, – con l'integrazione di *PostGIS* per la gestione di dati spaziali – e su *mod_python*. Il pacchetto è distribuito sotto licenza GPL ed è attualmente alla versione 1.0rc2.

### 4.2.1 Configurazione del servizio

Data l'evidente giovinezza del progetto, non esiste ancora un pacchetto pre-compilato e perciò bisogna procedere all'installazione del codice sorgente; ciò nonostante il processo risulta semplice ed immediato.

Dopo aver completato l'installazione si dovrà procedere alla configurazione del servizio, alla creazione del database ed al suo popolamento tramite *request* di *RegisterObservation* e *InsertObservation.* La configurazione consiste nell'apportare le opportune modifiche al file `sosSonfig.py` in cui risiedono variabili sul servizio, sul provider e le credenziali di connessione al database.

Per quanto riguarda la creazione del database, la procedura risulta semplice dato che insieme al software istSOS viene fornito un *schema* in sql già pronto, mentre per il suo popolamento la procedura è risultata più onerosa ed il procedimento seguito verrà illustrato nel paragrafo successivo.

### 4.2.2 Popolamento del database delle osservazioni

Per un corretto funzionamento del server SOS, i dati relativi alle centraline di rilevamento, ai sensori, e le osservazioni, devono essere inseriti tramite le operazioni standard di *RegisterObservation* e *InsertObservation*. Per compiere l'operazione di popolamento del database sono stati quindi realizzati due script: i file di testo con i dati (in formati CSV) vengono letti e, sulla base di questi, vengono generati i file XML necessari per le *request* di *registerObservation* e *insertObservation* che poi sono inviati al server SOS.

Il primo strumento è il file `RegisterSensors.py`, scritto in Python che si occupa di creare i documenti XML da inviare al servizio SOS per

la registrazione dei sensori. Il suo funzionamento si basa principalmente sui dati dei sensori definiti manualmente nel file `configSOS.py`: esso contiene dati sui sensori, sui fenomeni e sulle stazioni di rilevamento come ad esempio nomi, coordinate, unità di misura e gli ID assegnati a ciascuna entità dall'ARPA Lombardia. Dato che non vi è modo di estrapolare questo tipo di informazioni in maniera automatica, è risultato necessario definirle manualmente: ciò che fa lo script in questione è automatizzare la creazione delle *request* di *registerSensor* ed il loro invio al servizio.

Il secondo strumento (`insertObservation.py`) lavora in maniera molto simile ma lo scopo è quello di automatizzare la creazione di *request* di *insertObservation*. Il suo funzionamento, in questo caso, risulta essere decisamente più efficiente, dato che i dati relativi alle osservazioni sono prelevati in maniera del tutto automatica dai file CSV forniti dall'ARPA Lombardia. Lo script necessita soltanto di un parametro che definisce il *path* dove sono contenuti i file CSV da processare: partendo da questo parametro, lo strumento è in grado di leggere tutti i file CSV presenti nella cartella, estrarre da essi le informazioni delle osservazioni, ciascuna è caratterizzata dall'ID del sensore, dall'ora della misurazione e dal valore rilevato, inserirle in documenti XML per le *insertObservation* ed infine inviarli al server SOS.

Uno dei principali problemi che si è dovuto affrontare durante l'implementazione di quest'ultimo strumento riguarda la quantità di misurazioni inviate in una singola *insertObservation*. Come specificato anche dagli stessi sviluppatori sul sito di sviluppo del progetto (http://istgeo.ist.supsi.ch/projects/istSOS/wiki/ioissue), le specifiche SOS (OGC 06-009r6), non contemplano l'inserimento di un data array su una procedura dato che una *insertObservation* deve restituisce l'ID assegnato all'osservazione. Secondo invece quanto definito dallo

schema XML e da quanto testato personalmente l'invio di array di misurazioni risulta essere possibile. Per ovviare a tale problema si è deciso di dare la possibilità all'utilizzatore di poter scegliere quale tipo di approccio seguire. Se da un lato, l'inserimento di molteplici osservazioni con un un'unica richiesta di *insertObservation* non rispetta perfettamente gli standard, dall'altro è possibile ridurre drasticamente i tempi di esecuzione nel caso di grandi quantità di dati da inserire. Dopo qualche test infatti si è riscontrato che l'inserimento di un anno di osservazioni per un dato sensore (circa 8760 osservazioni) tramite una singola richiesta di *insertObservation* impiega poco meno di un secondo; se invece vengono effettuate 8760 differenti richieste, una per ogni osservazioni, i tempi di esecuzioni si aggirano intorno ai 10 minuti.

Il secondo problema che si è dovuto affrontare, è stato invece l'inserimento di osservazioni già presenti nel database. Per risolvere questo si è sfruttato un parametro non standard definito dagli sviluppatori dell'istSOS che permette di forzare la riscrittura delle osservazioni in un particolare istante o periodo di tempo. Per attivare questo parametro è stato aggiunto un parametro di input opzionale nello script.

Per quanto riguarda i valori considerati non validi, identificati da ARPA Lombardia con il valore -999, vengono regolarmente registrati nel database delle osservazioni, dato che comunque permettono di avere un'informazione più completa su una particolare osservazione.

## *4.3 pyWPS*

Il software ***pyWPS*** (http://pywps.wald.intevation.org/) è una

implementazione dello standard Web Processing Service OGC distribuito sotto licenza GPL e finanziata principalmente da HS-RS, società che opera nel settore del telerilevamento nella Repubblica Ceca e membro dell'OGC. L'applicazione è scritta completamente in Python e per questo motivo anche i *processi* dovranno essere implementati in questo linguaggio. Il vantaggio maggiore nell'utilizzare questo strumento è che è stato realizzato con supporto nativo a ***GRASS GIS*** (http://grass.itc.it/), applicazione open-source GIS, progetto ufficiale dell'*OSGeo* (*Open Source Geospatial Foundation*), utilizzata in tutto il mondo per la gestione e l'analisi di dati geo-spaziali, il processing di immagini, e la produzione e visualizzazione di carte in ambito accademico, commerciale così come in molte agenzie governative e private.

### 4.3.1 Configurazione del servizio

Dopo aver completato l'installazione, anche in questo caso tramite compilazione da codice sorgente, data la mancanza di pacchetti pre-compilati, bisogna procedere con la configurazione del servizio, apportando le opportune modifiche al file `pywps.cfg` in cui sono raccolte tutte le variabili relative al web server, al servizio WPS e a GRASS.

### 4.3.2 Processi implementati

Una volta configurato il servizio, il passo successivo consiste nell'implementazione dei *processi*. Come già accennato precedentemente, il server WPS è semplicemente un modello che permette la standardizzazione di processi ai fini della loro condivisione ed interoperabilità: il WPS non implementa perciò nessun processo o

funzionalità, ma semplicemente descrive come essi debbano essere sviluppati. Ogni processo dovrà perciò essere sviluppato seguendo alcuni criteri tra cui la necessità di definire alcuni metadati generici riguardo al processo stesso (un identificativo, un nome human-readable ed una descrizione dello stesso) e successivamente tutti gli input (necessari e opzionali) e gli output. Nei paragrafi successivi si vedrà nel dettaglio l'implementazione dei singoli processi.

Caratteristica da notare è che i dati delle osservazioni provengono dallo stesso database realizzato per il servizio SOS.

#### 4.3.2.1 Interpolation

Il processo implementato e definito *interpolation* permette al client di ottenere una mappa di interpolazione partendo dai valori di una data *observedProperty* di ciascuna centralina e dalle loro posizioni: tale immagine verrà sovrapposta alla carta in modo tale da poter valutare, la distribuzione spaziale di quel particolare inquinante sul territorio partendo da misurazione puntuali.

Il processo basa principalmente il suo funzionamento su GRASS, ed in particolare sulla funzione `v.surf.idw`, funzione che permette di ottenere una matrice raster di valori interpolati, generati da un set di dati puntuali distribuiti irregolarmente nello spazio, attraverso tecniche di approssimazione numerica (*weighted averaging*), in cui il valore di ciascuna cella viene calcolato dai valori dei punti più vicini in relazione alla distanza a cui questi punti si trovano dalla cella stessa.

Il processo si articola perciò in quattro differenti passi successivi:

1. vengono estratti dal database i valori e le coordinate di ciascun punto (sensori) e scritti in un file temporaneo. Con il comando `v.in.ascii` è stato poi possibile creare un layer (*mask*) vettoriale partendo dal file ASCII che verrà utilizzato come layer

di partenza per la generazione dell'interpolazione.

2. Il comando `g.region` ha permesso di definire la regione geografica di interesse, i cui limiti verranno considerati come estensione per generare l'interpolazione. La regione è stata definita in modo da includere tutti i punti definiti nel passo precedente aggiungendo un buffer di 1km: in questo modo è possibile valutare anche l'effetto dei punti che si trovano in prossimi degli estremi.
3. Dal layer vettoriale definito (*mask*) attraverso il comando `v.surf.idw`, precedentemente citato, si è generato un layer raster (*grid*) contente l'interpolazione dei punti.
4. Con il comando `r.colors` si è associata al raster una palette di colori ed infine con `r.out.png` si è creata l'immagine finale in formato PNG

#### 4.3.2.2 monthlyAverages

Questo processo offre la possibilità di calcolare le medie mensili per ciascuna proprietà osservata e per ciascuna centralina; il servizio restituisce i dati di un intero anno; questo per permettere l'integrazione con una funzionalità sviluppata nell'interfaccia (lato client) che facilita l'osservazione dell'evoluzione temporale delle medie (nell'arco di un anno) per mezzo di un comodo *slider*.

L'implementazione del processo risulta piuttosto semplice: per ogni mese viene effettuata una query al database in cui vengono estratte le medie di ciascun sensore in base alla proprietà osservata richiesta: i dati estratti vengono organizzati in un array il quale verrà restituito direttamente al client.

La caratteristica più rilevante di tale processo – oltre alla modalità con cui l'utente interagisce, ma che verrà illustrata nel capitolo

riguardante l'implementazione del client – è che la query non viene eseguita direttamente sul database delle osservazioni ma su una tabella intermedia in cui i dati riguardanti i valori medi mensili sono già stati calcolati e immagazzinati. Ovviamente ciò implica che il sistema o un amministratore debba regolarmente aggiornarla in base ai cambiamenti del database delle osservazioni.

Per calcolare i dati medi è stato implementato uno script che permette di eseguire questo aggiornamento in maniera del tutto automatica. Lo script non fa altro che andare a recuperare tutti i dati delle osservazioni, calcolare le medie mensili e salvarle nella tabella in questione. La peculiarità di questo script è che è stato implementato anche con lo scopo di permettere la rigenerazione dei suddetti dati: nel caso in cui nella tabella delle medie siano già presenti dati riguardanti una data proprietà osservata in un determinato periodo, questi verranno completamente ricalcolati e sostituiti. Ciò è stato possibile anche grazie alla definizione di quattro diversi parametri di input (tutti opzionali) che permettono di restringere il campo di esecuzione del processo, in modo tale da non dover rigenerare tutto il database ogni volta – il che può risultare decisamente oneroso. I possibili input accettati dallo script sono i seguenti:

- *observedProperty*: questo parametro permette, nel caso in cui sia definito, di restringere l'esecuzione dello script solo ad una data proprietà osservata. Perciò, nel caso cui essa non venga definita, l'aggiornamento riguarderà tutte le proprietà osservate definite nel sistema.
- *foi*: questo parametro restringe invece il campo di esecuzione dello script solo ad una *featureOfInterest*. Potrebbe rivelarsi particolarmente utile nel caso in cui vi sia necessità di rigenerare i dati di una sola centralina, magari a causa di un guasto o di un

malfunzionamento.

- *start* & *end*: questi due parametri permettono invece di definire dei limiti temporali al processo di calcolo delle medie. Nel caso essi non vengano definiti, verranno interrogati e ricalcolati tutti i dati disponibili nel database.

Pur rimanendo tutti parametri opzionali, in modo da consentire una eventuale rigenerazione dell'intero database, il loro utilizzo è altamente consigliato, soprattutto per quanto riguarda l'aspetto temporale.

Una caratteristica di particolare rilevanza è l'automatica esclusione – da parte della porzione di codice dello script che calcola le medie – dei valori considerati non validi identificati dal numero -999.

# 5 Implementazione dell'interfaccia

## *5.1 Tecnologie utilizzate*

La realizzazione dell'applicazione, oltre ai già descritti servizi geografici SOS e WPS con tutti i software e le librerie necessarie per la loro implementazione, ha incluso anche l'utilizzo di altre tecnologie atte a sviluppare una interfaccia che permettesse all'utente di interagire in maniera soddisfacente con i servizi. Tali tecnologie includono ***MapServer***, software in grado di generare mappe ed ***OpenLayers***, framework in grado di fornire tutti i controlli necessari per la navigazione delle mappe. Il linguaggio lato server utilizzato per l'implementazione è *PHP*, mentre le interazioni lato client sono state realizzate grazie a Javascript, ed al framework *jQuery*.

### 5.1.1 MapServer

MapServer (http://mapserver.org/) è una piattaforma open source per la pubblicazione di dati spaziali e mappe interattive attraverso il web. Il suo sviluppo, iniziato quasi 15 anni fa presso l'Università del Minnesota, ha raggiunto livelli di stabilità e maturità che lo hanno portato ad essere uno dei principali progetti supportati dall'*Open Source Geospatial Foundation* (*OSGeo*) ed uno dei principali software

open-source per il web mapping.

Il suo funzionamento è principalmente basato sui *Mapfile*, file testuali che includono tutti i parametri e le variabili necessari per poter generare un'immagine: il *mapfile* contiene ad esempio informazioni sui layer da generare, sui dati da utilizzare (che possono essere sia vettoriali che raster), sui colori e sulle etichette, sulle eventuali legende, barre di scala, ecc...). Vi sono poi due differenti approcci che possono essere utilizzati per implementare una sua istanza: la prima, e la più semplice consiste in un CGI in grado di generare un'immagine quando riceve una richiesta; la seconda, utilizzata nel sistema realizzato, permette invece di sfruttare le potenzialità di un linguaggio dinamico attraverso le librerie *Mapscript*, che possiedono interfacce per diversi linguaggi tra cui PHP e Python.

MapServer comunque, permette solo di generare immagini, e non fornisce alcuno strumento avanzato che permetta di navigarle e gestirle: tale compito è stato affidato ad *OpenLayers*, che verrà illustrato nel prossimo paragrafo.

### 5.1.2 OpenLayers

OpenLayers (http://openlayers.org/) consiste in un framework, interamente scritto in Javascript, che dà la possibilità di inserire mappe navigabili in qualsiasi pagina web. É uno strumento che lavora completamente lato client e fornisce un'interfaccia in stile *Google-Maps*, che permette di interagire con la mappa in maniera molto intuitiva e veloce.

Il successo che OpenLayers ha avuto e che lo ha portato ad essere uno strumento molto utilizzato, viene non solo dalla sua efficienza nell'utilizzo, ma soprattutto dalla semplicità della sua interfaccia, che

non vincola in alcun modo lo sviluppatore a modificare il *layout* delle proprie pagine per poterlo implementare: esso consiste semplicemente nell'immagine della mappa stessa (le cui dimensioni possono essere adattate a piacimento) in cui tutti i controlli sono stati integrati. Nella figura 6 è possibile osservare i principali controlli per la navigazione della mappa.

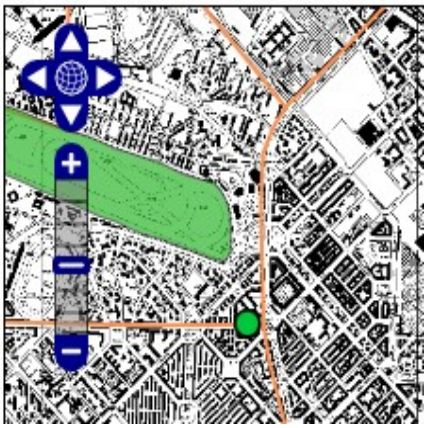

***Figura 6**: Dettaglio dei controlli di OpenLayers*

OpenLayers è in grado di creare mappe navigabili sovrapponendo tra di loro differenti *layers:* oltre a supportare svariati formati di *layers*, tra cui semplici immagini, oppure mappe provenienti da server WMS, WFS, oppure ma MapServer, da ArcIMS, o anche da servizi come GooleMaps, NASA WorldWind, o Yahoo.

## *5.2 Implementazione del client*

### 5.2.1 Descrizione dell'interfaccia

Gli obiettivi primari che si è cercato di raggiungere durante la creazione e l'implementazione dell'interfaccia, sono stati principalmente semplicità e praticità. È soprattutto per questo motivo

che essa appare piuttosto spoglia; ma questa aspetto minimale scaturisce soltanto dalla volontà di ottenere un sistema in grado di adattarsi a qualunque tipo di interfaccia pre-esistente senza vincolare lo sviluppatore in alcun modo. Con questo stesso spirito sono stati anche creati tutti i controlli che permettono all'utente di interagire con il sistema, ed in particolare con i servizi geografici SOS e WPS. Il modo in cui i controlli vengono gestiti e l'interazione con l'utente rimangono infatti completamente indipendenti da ciò che effettivamente svolge le funzioni più cruciali dell'applicazione (che verranno descritte nel dettaglio nei paragrafi successivi), e cioè la comunicazione con le istanze dei geoservizi. Tutto ciò si traduce nella possibilità per lo sviluppatore di definire l'interfaccia per i controlli ai geoservizi a proprio piacimento, concentrandosi soltanto sulla gestione degli input dell'utente.

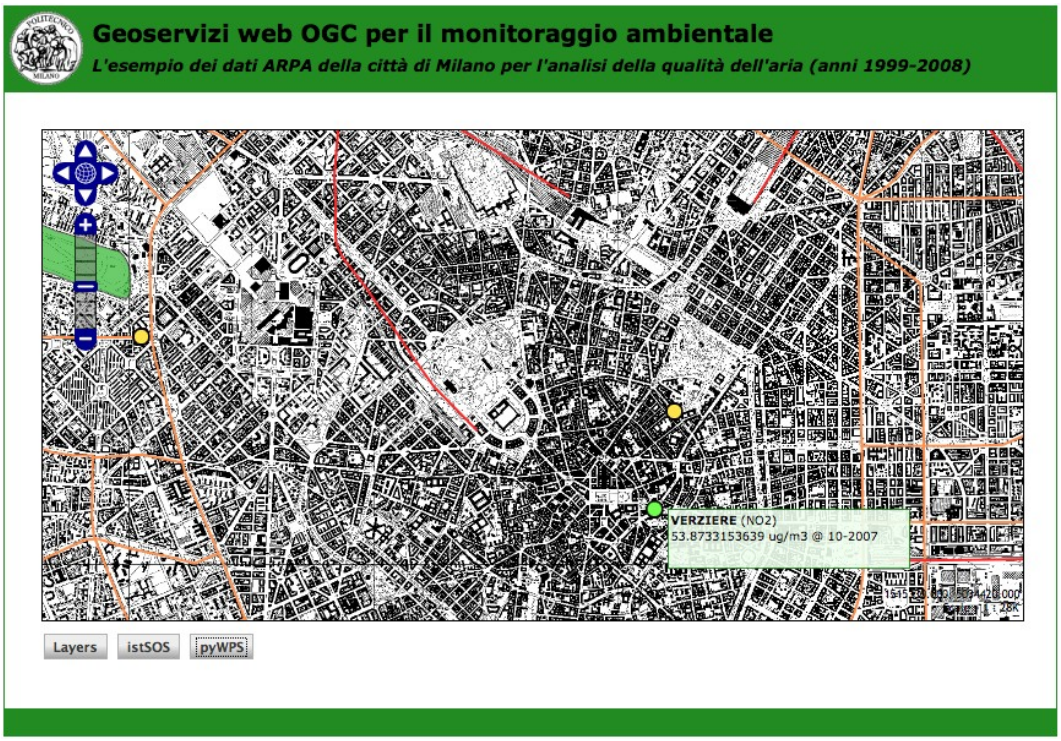


***Figura 7**: Interfaccia completa*

Come si può osservare nell'illustrazione precedente, l'interfaccia realizzata consiste essenzialmente in una grande mappa, sotto la quale sono stati collocati alcuni strumenti per la gestione dei layer ed un menu che permette di accedere ai controlli dei geoservizi. Tutti i controlli per la navigazione della mappa offerti da OpenLayers risultano quindi – come precedentemente osservato – essere completamente integrati in essa.

#### 5.2.1.1 I markers delle osservazioni

Un altro elemento offerto da OpenLayers che è stato sfruttato per la visualizzazione dei risultati ottenuti dai servizi SOS e WPS è il ***marker***. Questo tipo di oggetto consiste in una semplice icona che permette di indicare sulla mappa l'esatta posizione di un qualche tipo di entità. Questo oggetto si è perciò rivelato particolarmente utile per la visualizzazione delle centraline di rilevamento e delle relative osservazioni. É stata generata infatti una scala di 12 diversi *markers* (Figura 8), con variazioni graduali di colori da verde al rosso, utilizzati per evidenziare il valore della misurazione di una particolare osservazione: a questi è stato aggiunto poi un *marker* di colore nero per la visualizzazione di osservazioni con misurazioni non valide o fuori scala. Grazie a OpenLayers è inoltre possibile associare ad ogni *marker* un pop-up che, comparendo al click dell'utente è in grado di fornire informazioni aggiuntive.

***Figura 8**: Markers delle osservazioni*

### 5.2.2 Implementazione del client SOS

Il lavoro svolto per implementare il client per il servizio SOS ha

rivolto la propria attenzione principalmente nel trovare la maniera più semplice possibile di consentire ad un utente di interagire con le richieste di tipo *getObservation*. Si è così realizzato un controllo (Figura 9) che permette all'utente, dopo aver selezionato il fenomeno a cui si è interessati e l'anno, di sfruttare un comodo *slider* per selezionare l'istante di cui si vogliono ottenere le osservazioni. Una volta effettuata la selezione dei *markers* verranno disegnati sulla mappa in corrispondenza di tutti sensori in cui esiste una osservazione.


***Figura 9**: Dettaglio del client SOS*

Per gestire la suddetta interazione, è stato necessario implementare tre differenti classi:

- `OpenLayers.istSOS`: definita nel file `client.js`, gestisce tutto il processo di invio della richiesta, da quando l'utente seleziona i parametri a quando, dopo aver ricevuto la *response* della *getObservation* dal server SOS, genera un nuovo *layer* di *features* da aggiungere alla mappa. Il layer di tipo *features* non esiste e perciò e stato necessario implementarlo.
- `OpenLayers.Layer.Features.istSOS`: questa classe, definita nel file `layer.Features.istSOS.js`, permette di generare un layer di oggetti *Feature*, cioè di oggetti che includono un *marker* ed un pop-up: ciò che genererà sarà quindi un layer composto da una serie di marker, che identificano i sensori, ognuno dei quali selezionabile per ottenere informazioni sulla centralina, sulla proprietà osservata e, ovviamente, sull'osservazione. Questa classe eredita le componenti principali da un'altra classe

implementata, `OpenLayers.Layer.Features`, estendendone le caratteristiche per adattarsi all'output fornito dalla *response* della *getObservation*.

- `OpenLayers.Format.istSOSGetObservation`: implementata nel file `format.istSOS.GetObservation.js`, definisce il formato del file XML restituito dalla *getObservation* in modo tale che il parser XML sia in grado di leggerlo ed estrapolare le informazioni come ad esempio la *featureOfInterest*, la *observedProperty,* ed il valore dell'osservazione.

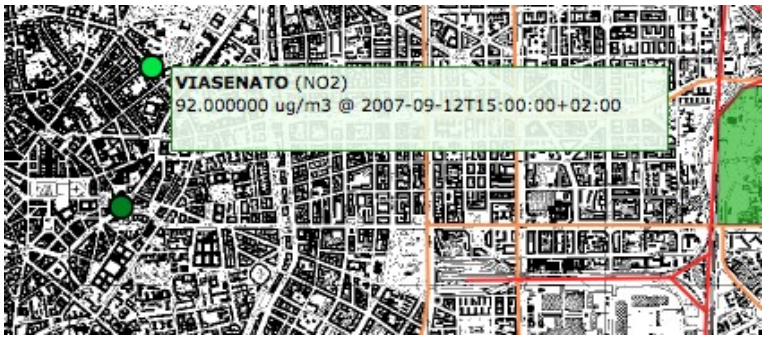


***Figura 10**: Feaures (markers + pop-ups) delle osservazioni*

### 5.2.3 Implementazione del client WPS

In maniera del tutto simile a quanto fatto per il client del servizio SOS, si è svolta l'attività per l'implementazione dell'interfaccia per il servizio WPS, realizzando due differenti *form* di interazione, uno per ciascuna delle operazioni offerte dal servizio.

Per l'operazione denominata *monthlyAverages* è stato progettato uno *slider* che permettesse di creare una sorta di animazione volta alla valutazione del trend delle media mensili delle osservazioni nell'arco di un anno. L'utente, dopo aver selezionato l'anno ed il fenomeno a cui è interessato, effettua la richiesta al server WPS al termine della quale

ottiene uno slider che permette la selezione del mese di cui visualizzare sulla mappa le relative medie mensili. La caratteristica di maggior pregio riguarda proprio questo *slider*, dato che trascinando il cursore da un mese all'altro i *markers* verranno automaticamente aggiornati in base al mese corrente senza la necessità di dover effettuare una richiesta al server WPS (Figura 11): questa infatti, anche nelle migliori condizioni, non sarebbe comunque sufficientemente veloce da poter aggiornare il layer delle *features* durante il trascinamento dello slider.

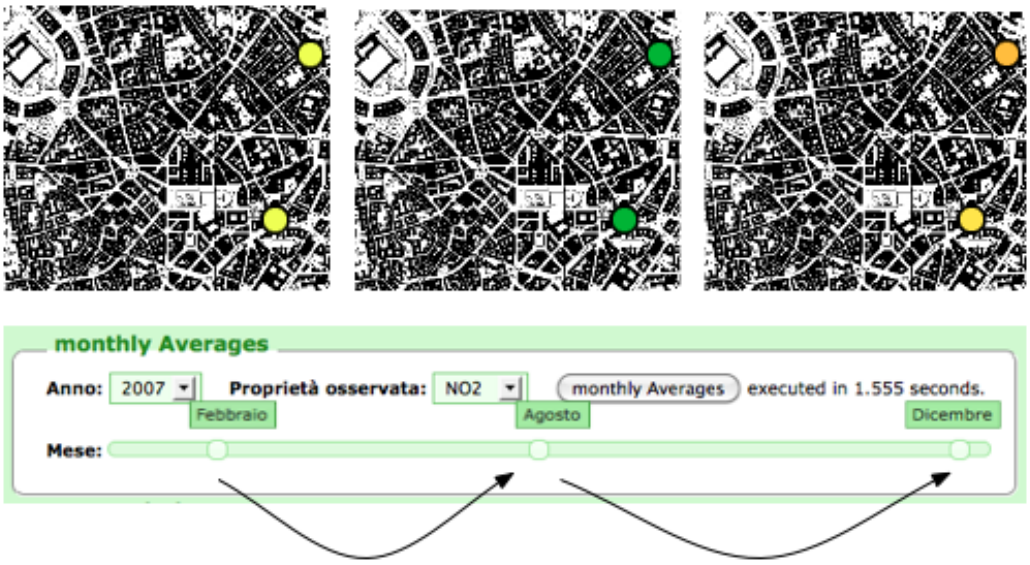


***Figura 11**: Slider di selezione del mese relativo alle medie mensili.*

L'operazione di *interpolation*, invece, è stata implementata realizzando un form di selezione dell'istante di cui ottenere l'interpolazione, del tutto simile a quello del client SOS: ad avvenuta richiesta l'utente ottiene l'immagine dell'interpolazione direttamente sovrapposta alla mappa corrente (Figura 12), con un grado di trasparenza tale da poter permettere di vedere comunque gli altri layer visualizzati.

***Figura 12**: Esempio di interpolazione*

Anche per l'implementazione del client WPS è stato necessario creare delle classi che permettessero di espandere le funzionalità di OpenLayers.

- `OpenLayers.pyWPS`: definita nel file `client.pyWPS.js`, è una classe in grado di gestire tutto il processo di invio delle richieste al servizio WPS e di ricezione delle *response* con la successiva visualizzazione degli output. Nel caso dell'interpolazione l'output sarà un layer di tipo *Image*, mentre per l'operazione *monthlyAverages* sarà una pila di layer *Features;*
- `OpenLayers.Layer.Features.pyWPS`: così come per la il servizio SOS anche il WPS prevede la visualizzazione delle stazioni di rilevamento tramite *features*; è stato così necessario implementare la creazione di una classe – anch'essa che estende le funzioni della classe `OpenLayers.Layer.Features` – (definita nel file `layer.Features.pyWPS.js`) in grado di generarne il relativo *layer*;
- `OpenLayers.Format.WPS`: similmente a quanto fatto per il SOS, anche il WPS ha necessitato di una classe (definita nel file

`format.WPS.js`) in grado di effettuare il parser del documento restituito dall'operazione di *Execute*, dal server WPS. Questa classe è stata realizzata sfruttando parte del codice di un prototipo di supporto allo standard WPS, realizzato da uno sviluppatore di OpenLayers, e disponibile sul sito di sviluppo (http://trac.openlayers.org/browser/sandbox/august/openlayers/2.8%2B/lib/OpenLayers/Format/WPS/v1.js?rev=10263).

# 6 Test prestazionali

Uno dei principali motivi per cui sono stati scelti i software *istSOS* e *pyWPS* per implementare rispettivamente i geoservizi SOS e WPS, è stato sicuramente l'elevato grado di semplicità con cui questi software sono stati realizzati. E proprio questa semplicità nella struttura e nell'organizzazione del software ha fatto supporre una conseguente velocità nell'esecuzione delle richieste. Per tale motivo sono stati effettuati dei test prestazionali in grado di sollecitare e sottoporre ad un elevato carico di lavoro i due rispettivi servizi in modo tale da poterne giudicare le prestazioni generali.

Per fare questo è stato utilizzato il software ***Apache JMeter*** che verrà descritto nel prossimo paragrafo.

## *6.1 Apache JMeter*

*Apache JMeter* (http://jakarta.apache.org/jmeter/) è una applicazione desktop sviluppata da *Apache Software Foundation* per eseguire test di carico sia su applicazioni web che desktop che permettano di valutare il livello di prestazioni. Può essere utilizzato per simulare un sovraccarico su un server, su una rete o su una singola risorsa per analizzare la robustezza ed il livello di prestazioni a differenti livelli di

carico. É in grado di interagire sia con risorse statiche che dinamiche come ad esempio files, CGI, Servlets, Scripts, e web services, ed è in grado di gestire richieste contemporanee grazie al supporto al multithreading.

## 6.2 Configurazioni dei test

Per ogni risorsa che si vuole testare con JMeter, è possibile, oltre a definire la risorsa stessa con tutti i relativi input, impostare tre differenti variabili che andranno ad identificare la quantità di carico a cui si vorrà sottoporre tale risorsa:

- *Number of Threads*: questa variabile indica il numero di *thread* concorrenti (utenti) che verranno lanciati
- *Ramp-Up Period*: questa variabile identifica quanto tempo JMeter dovrà impiegare per lanciare tutti i thread definiti con la variabile precedente. JMeter lancerà ciascun thread uno dopo l'altro, aspettando un tempo pari al rapporto tra il ramp-up ed il numero di thread definito: se ad esempio vengono definiti 10 thread ed un ramp-up di 100 secondi, verrà lanciato un thread ogni 10 secondi (100/10).
- *Loop Count*: definisce il numero di volte che il test verrà eseguito.

Per semplicità, si è deciso di definire un valore fisso per il Ramp-Up a 60 secondi ed per il Loop Count a 1, mentre un valore variabile per il numero di threads da lanciare in modo tale da poter valutare il cambiamento delle prestazioni al variare del carico di richieste.

Ogni test è stato eseguito con il processore in *idle* e nessun altro

processo attivo. È stato effettuato inoltre attraverso una rete domestica, configurando il software JMeter su una macchina diversa da quella in cui risiedevano i servizi SOS e WPS in modo tale che si evitasse che JMeter stesso potesse in qualche modo influire sui risultati. Tra un test e l'altro inoltre, si è sempre atteso un tempo sufficiente da riportare il processore ad uno stato di riposo.

## *6.3 Test realizzati*

### 6.3.1 istSOS

Il test sull'istanza del server SOS è stato effettuato predisponendo l'esecuzione di richieste di tipo *getObservation* da eseguire su tutte e 8 le differenti proprietà osservate nello stesso istante temporale: si è reputato infatti che il variare dell'istante delle osservazioni, purché valido per tutte le proprietà osservate, non potesse in alcun modo incidere sui tempi di esecuzione della richiesta.

Per l'esecuzione dei test si è partiti fissando il numero di thread a 60 in modo tale da effettuare una richiesta ogni secondo e, ottenendo un valore di circa 190 millisecondi, si è deciso di continuare con i test successivi incrementando gradualmente il numero di thread da eseguire.

| n° threads | requests/s | average [ms] | median [ms] | 90% line [ms] |
|---|---|---|---|---|
| 60 | 1,0 | 178 | 177 | 223 |
| 100 | 1,7 | 187 | 177 | 224 |
| 200 | 3,3 | 184 | 174 | 221 |
| 300 | 5,0 | 178 | 165 | 255 |
| 400 | 6,7 | 165 | 153 | 264 |
| 500 | 8,3 | 141 | 121 | 224 |
| 600 | 10,0 | 126 | 103 | 196 |
| 700 | 11,7 | 123 | 109 | 166 |
| 800 | 13,3 | 123 | 108 | 151 |
| 900 | 15,0 | 131 | 120 | 179 |
| 1000 | 16,7 | 140 | 129 | 204 |
| 1100 | 18,3 | 178 | 152 | 292 |
| 1200 | 20,0 | 2166 | 288 | 8530 |

***Tabella 3**: Risultati dei test prestazionali del servizio SOS effettuati con JMeter*

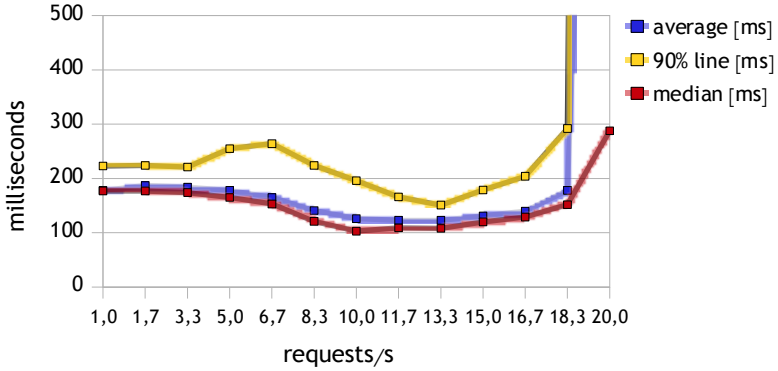


***Figura 13:** Risultati dei test prestazionali del servizio SOS effettuati con JMeter*

Dai risultati ottenuti (Tabella 3 e Figura 13) si osserva che fino a 1100 thread al minuto, che corrispondono a circa 18 richieste al secondo, il server SOS riesce a soddisfare senza alcun problema tutte le richieste e le porta a termine con tempi che oscillano tra i 150 e 200 millisecondi. Oltre le 18 richieste al secondo, i ritmo con cui vengono inviate al server risulta essere maggiore rispetto a quello impiegato per elaborarle: il carico di lavoro così si accumula ed i tempi di risposta aumentano esponenzialmente.

### 6.3.2 pyWPS

L'esecuzione dei test sul servizio WPS si sono svolti in maniera del tutto simile ai test eseguiti sul servizio SOS, inviando richieste di tipo *Execute*. A differenze del software istSOS, il pyWPS include un meccanismo per limitare il numero massimo di connessioni possibili impostando tale valore nel file di configurazione. Nel caso in cui vengano eseguite un numero di richieste maggiore a quello consentito, queste verranno bloccate, lanciando un'eccezione e restituendo al client un documento XML contenente un messaggio di errore.

Il documento di errore restituito risulta però essere, in termini di comunicazione HTTP, una *response* del tutto valida e quindi JMeter non è in grado di distinguere tra una richiesta di *Execute* perfettamente eseguita oppure una bloccata da tale sistema. Si è deciso quindi di fissare il limite ad un valore sufficientemente alto, ad esempio a 100, in modo da poter eseguire i test senza che questi potesse influire in alcun modo sui risultati dato che il tempo di ricezione di una richiesta *Execute* correttamente eseguita non è in alcun modo comparabile ai tempi di ricezione del messaggio di errore.

É doveroso comunque ricordare che nel caso del servizio WPS, i tempi con cui una richiesta può essere eseguita non dipendono da quanto il software pyWPS sia performante, ma principalmente da quanto ciascun *process* sia oneroso in termini di potenza computazionale. Per questo motivo sono stati effettuati test differenti per i due processi e come ci si poteva attendere, sono stati ottenuti risultati anche piuttosto differenti.

#### 6.3.2.1 Interpolation

In questo test è stata esclusa come possibile input la proprietà osservata SO2 (Biossido di zolfo) in quanto presenti dati di un solo sensore e quindi non sufficienti per eseguire il processo.

| *n° threads* | *requests/s* | *average [ms]* | *median [ms]* | *90% line [ms]* |
|---|---|---|---|---|
| *30* | 0,5 | 1076 | 1074 | 1184 |
| *40* | 0,67 | 1080 | 1072 | 1180 |
| *50* | 0,83 | 1093 | 1095 | 1205 |
| *60* | 1 | 1050 | 1014 | 1209 |
| *70* | 1,17 | 1061 | 1046 | 1193 |
| *80* | 1,33 | 1186 | 1185 | 1337 |
| *90* | 1,5 | 1217 | 1222 | 1383 |
| *100* | 1,67 | 1276 | 1250 | 1523 |
| *110* | 1,83 | 1484 | 1492 | 1714 |
| *120* | 2 | 1829 | 1846 | 2210 |
| *130* | 2,17 | 2755 | 2760 | 3429 |
| *140* | 2,33 | 4786 | 4948 | 6589 |
| *150* | 2,5 | 8714 | 8940 | 13416 |

***Tabella 4**: Risultati dei test prestazionali del processo **interpolation** (pyWPS) effettuati con JMeter*

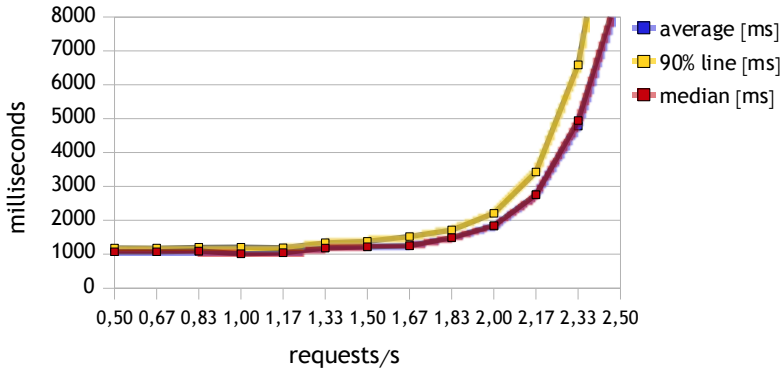


***Figura 14**: Risultati dei test prestazionali del processo interpolation (pyWPS) effettuati con JMeter*

A differenza del servizio SOS, questo processo WPS risulta essere notevolmente più oneroso in termini computazionali e perciò si ottiene un tempo medio di esecuzione compreso tra 1,5 e 2 secondi (Tabella 4 e Figura 14); il fattore determinante è che comunque queste

prestazioni sono state registrate soltanto per carico inferiore alle 2 richieste al secondo. Per carichi maggiori si registra un aumento esponenziale dei tempi.

#### 6.3.2.2 monhtlyAverages

Per questo test, non sono state adottati particolari accorgimenti oltre a quelli già descritti per i servizi precedenti.

| *n° threads* | *requests/s* | *average [ms]* | *median [ms]* | *90% line [ms]* |
|---|---|---|---|---|
| *10* | 0,17 | 1867 | 1868 | 1926 |
| *20* | 0,33 | 1945 | 1861 | 2493 |
| *30* | 0,50 | 1922 | 1879 | 1992 |
| *40* | 0,67 | 2101 | 2083 | 2325 |
| *50* | 0,83 | 2471 | 2410 | 2866 |
| *60* | 1,00 | 2696 | 2702 | 3011 |
| *70* | 1,17 | 4198 | 4280 | 5066 |
| *80* | 1,33 | 10692 | 11393 | 15604 |

***Tabella 5**: Risultati dei test prestazionali del processo **monthlyAverages** (pyWPS) effettuati con JMeter*

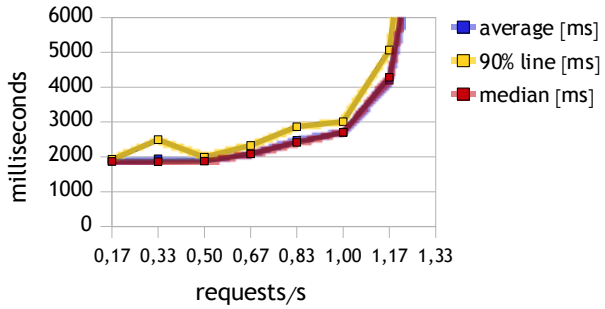


***Figura 15**: Risultati dei test prestazionali del processo **monthlyAverages** (pyWPS) effettuati con JMeter*

Dai risultati ottenuti (Tabella 5 e Figura 15) si osserva come questo processo sia ancora più oneroso del precedente: impiega dai 2 a 2,5 secondi per effettuare una richiesta, e non è in grado di andare oltre ad una richiesta al secondo senza andare in overload.

## 6.4 Conclusioni sui test effettuati

Per poter giudicare l'assoluta bontà dei dati raccolti sarebbe opportuno effettuare un maggior numero di test, anche su macchine differenti, ma soprattutto implementando i due servizi con tecnologie differenti, in modo tale da aver un termine di paragone con cui confrontare i suddetti risultati.

In ogni caso sono principalmente due i fattori che fanno ragionevolmente ritenere le prestazioni ottenute sufficienti per una possibile applicazione del sistema realizzato.

Innanzitutto il fatto che la macchina su cui sono stati effettuati i test risulta essere non di ultima generazione: è stato infatti utilizzato un normale PC desktop con processore AMD Athlon 64 X2 3800+ operante alla frequenza di 2000MHz e rilasciato ormai più di 4 anni fa (Maggio 2006) e corredato da 4Gb di memoria RAM. Ciò fa presupporre che, data la impressionante velocità con cui la tecnologia si evolve, con una macchina più recente sia possibile ottenere risultati decisamente migliori.

In secondo luogo, si dovrebbe considerare il fatto che, questo tipo di applicazioni, nonostante l'ampio margine di crescita, rimangono comunque appartenenti ad un settore piuttosto di nicchia, e molto probabilmente non sarà necessario pianificare l'implementazione di una macchina in grado di supportare un numero estremamente elevato di richieste contemporanee.

Per quanto riguarda, invece, il numero massimo di connessioni consentite che può essere impostato nel software pyWPS, è necessario notare che la sua corretta configurazione dipende soprattutto da che cosa si ritenga prioritario nell'implementazione di tale servizio. Nel caso si decida di assegnare un valore relativamente basso, si assicura

che tutte le richieste vengano eseguite in tempi ragionevoli; d'altro canto alcune di esse non verranno affatto considerate obbligando il client a ritentare in un secondo momento. Nel caso, invece, si decida di assegnare un valore molto alto, si assicura che tutte le richieste vengano in qualche modo portate a termine a discapito però dei tempi di esecuzione, che in caso di sovraccarico potrebbero diventare anche molto lunghi. Bisogna inoltre tener conto che il valore di operazioni massime possibili viene definito a livello di server WPS e non a livello di ciascun processo: ne consegue quindi che un determinato valore possa essere ideale per un processo ma non per un altro; ciò obbliga perciò a dover valutare tale limite in maniera da adattarsi ottimalmente a tutti i processi implementati.

# 7 Conclusioni

## 7.1 Criticità riscontrate

Sono principalmente due le criticità riscontrate durante le diverse fasi di implementazione del sistema dei servizi geografici OGC *Sensor Observation Service* e *Web Processing Service*.

La prima riguarda una limitazione intrinseca delle tecnologie utilizzate, ed in particolare il fatto che il software *istSOS*, utilizzato per l'implementazione dl servizio SOS, imponga una ben definita struttura da adottare per il database delle osservazioni. Da una parte ciò può essere visto come un aspetto positivo, in quanto solleva lo sviluppatore dal delicato ed oneroso compito di modellazione del database ed inoltre assicura un ben specifico grado di aderenza alle specifiche del geoservizio; d'altro canto però, come già illustrato nei capitoli precedenti, pone una notevole limitazione alle libertà dello sviluppatore nell'organizzare i dati come meglio crede. Si pensi poi alle difficoltà che possono scaturire dalla necessità di integrazione con un sistema già operativo in cui si voglia aggiungere la capacità di offrire questo determinato servizio – come può essere lo stesso sistema ARPA – e che ovviamente sia già dotato di un proprio database con immagazzinati i dati delle osservazioni.

L'altro punto critico riscontrato riguarda invece il dominio di applicazione di questo elaborato ed in particolare una limitazione imposta dal tipo di dati attualmente resi disponibili dalla regione Lombardia. Se da un lato sembra vi siano segnali di rinnovamento, dall'altro i dati ed in particolare i dati raster, come carte tecniche regionali, orto-foto, etc.., attualmente resi disponibili dalla Regione Lombardia attraverso il suo portale cartografico, sono caratterizzati dalla proiezione *Gauss-Boaga* (EPSG:3003). Ciò obbliga, nel caso in cui si vogliano utilizzare i suddetti dati raster, a mantenere la medesima proiezione cartografica in tutto il GIS per la visualizzazione delle mappe e delle osservazioni (anch'esse in *Gauss-Boaga*) e perciò impedisce l'importazione di layer WMS esterni (come ad esempio Google-Maps) che, per la maggior parte, sono disponibili in *WGS 84* (EPSG:4326).

## *7.2 Pregi e sviluppi futuri*

Nonostante i problemi incontrati, le tecnologie implementate sono comunque risultate funzionali e ben progettate. Sicuramente la semplicità nella configurazione e pochi requisiti necessari per l'installazione dei differenti software hanno giocato un ruolo chiave nella loro effettiva robustezza e nelle loro prestazioni. Un altro aspetto che è subito apparso un notevole vantaggio è sicuramente il linguaggio con cui queste tecnologie sono state implementate, con particolare rilievo il software pyWPS: essendo Pyhton un linguaggio pseudo-interpretato (come ad esempio il php) non necessita di essere compilato e facilita decisamente una eventuale modifica al codice. Negli ultimi anni poi ha potuto assistere ad una decisa fase di sviluppo che ha portato alla creazione di innumerevoli librerie e moduli

aggiuntivi che ne hanno esteso le potenzialità anche in campo della gestione e manipolazione di dati geografici.

Il lavoro svolto in questo progetto si è posto comunque come obiettivo quello di identificare una possibile applicazione e mostrare alcune delle possibilità offerte dai servizi geografici implementati; si è trattato di una implementazione di base, la quale potrebbe senza dubbio essere resa più funzionale, migliorando e potenziando le possibili interazione dell'utente con il sistema.

Il servizio SOS è apparso uno standard completo e funzionale nella sua parte riguardante l'implementazione del server del servizio; risulta invece, molto probabilmente perché ancora piuttosto giovane, decisamente carente per quanto riguarda le implementazioni della parte relativa al client. É quindi su questo che si dovrebbe cercare di indirizzare gli sforzi: in questo modo si potrebbe riuscire effettivamente a realizzare un servizio completo che permetta di sfruttarne appieno tutte le potenzialità.

Notevoli potenzialità sono state riscontrate anche nel servizio WPS che, offrendo soltanto il contenitore, e gli strumenti necessari per la realizzazione di processi, risulta essere in grado di permettere l'implementazione di una infinita serie di possibili servizi, come ad esempio il calcolo di statistiche avanzate, di correlazioni tra differenti fenomeni, la creazione di grafici per l'osservazione dei trend, temporali, etc...

# 8 Bibliografia e Sitografia

Doyle A. & Reed C. (2001). Introduction to OGC Web Services. OGC Whitepaper.

IBM Developer Works: Standard and Web services. http://www.ibm.com/developerworks/webservices/standards/. Ultimo accesso il 22/06/2010

W3C: Web Services Architecture. http://www.w3.org/TR/ws-arch/. Ultimo accesso il 22/06/2010

Open Geospatial Consortium. http://www.opengeospatial.org. Ultimo accesso il 23/06/2010

de la Beaujardiere J. (2006). OpenGIS® Web Map Server Implementation Specification, OpenGIS® Implementation Specification, OGC, Document Number: 06-042.

Vretanos P. A. (2005). Web Feature Service Implementation Specification, OpenGIS® Implementation Specification, OGC, Document Number: 04-094.

Whiteside A., Evans D. J. (2008). Web Coverage Service (WCS) Implementation Standard, OpenGIS® Implementation Standard, OGC, Document Number: 07-067r5.

Schut P. (2007). OpenGIS® Web Processing Service, OpenGIS® Standard, OGC, Document Number: 05-007r7.

Botts M., G. Percivall, C. Reed & J. Davidson (2007). OGC Sensor Web Enablement: Overview and High Level Architecture. OGC Whitepaper. OGC, Document Number: 07-165.

Na A. & Priest M. (2007). Sensor Observation Service, OpenGIS® Implementation Standard, OGC, Document Number: 06-009r6.

OGC Network: Sensor Observation Service. http://www.ogcnetwork.net/SOS. Ultimo accesso il 02/07/2010

Integrated Ocean Observing System (IOOS®) http://ioos.gov/. Ultimo accesso il 01/07/2010

NDBC (National Data Buoy Center) SOS Server (NOAA) http://sdf.ndbc.noaa.gov/. Ultimo accesso il 25/06/2010

Istituto Scienze della Terra: HydroPortal. http://istgeo.ist.supsi.ch/site/hydroPortal. Ultimo accesso il 26/06/2010

52°North Sensor Observation Service http://52north.org/SensorWeb/sos/index.html. Ultimo accesso il 25/07/2010

OGC: Registered Products. http://www.opengeospatial.org/resource/products. Ultimo accesso il 04/07/2010

Degree3. Onine at: http://www.deegree.org/. Ultimo accesso il 26/06/2010

52°North: 52°n WPS. http://52north.org/maven/project-sites/wps/52n-wps-webapp/. Ultimo accesso il 26/06/2010

ERDAS APOLLO Professional. http://www.erdas.com/Products/ERDASProductInformation/tabid/84/currentid/3148/objectid/3148/default.aspx. Ultimo accesso il 04/07/2010

ARPA Lombardia: Qualità dell'Aria. http://ita.arpalombardia.it/ITA/qaria/Home.asp. Ultimo accesso il 28/06/2010

ARPA Lombardia: Archivio dei dati rilevati. http://ita.arpalombardia.it/ITA/qaria/doc_RichiestaDati.asp. Ultimo accesso il 28/06/2010

GEOportale della Lombardia. http://www.cartografia.regione.lombardia.it/geoportale. Ultimo accesso il 01/07/2010

istSOS Documentation: http://istgeo.ist.supsi.ch/software/istsos/documentation/index.html. Ultimo accesso il 29/06/2010

istSOS: Development Site. http://istgeo.ist.supsi.ch/projects/istSOS/. Ultimo accesso il 29/06/2010

PyWPS 3.2.0 Documentation: http://pywps.wald.intevation.org/documentation/pywps-3.2/. Ultimo accesso il 29/06/2010

PyWPS Course v3.x documentation: http://pywps.wald.intevation.org/documentation/course/. Ultimo accesso il 29/06/2010

Grass GIS. Online at http://grass.itc.it/. Ultimo accesso il 26/07/2010

GRASS GIS: GRASS GIS 6.4 Manual Reference. Online at http://grass.itc.it/grass64/manuals/html64_user/index.html. Ultimo accesso il 26/07/2010

MapServer. http://mapserver.org/. Ultimo accesso il 30/06/2010

OpenLayers. http://openlayers.org/. Ultimo accesso il 30/06/2010

OpenLayers: Documentation. http://docs.openlayers.org/. Ultimo accesso il 29/06/2010

OpenLayers: JavaScript API Documentation. http://dev.openlayers.org/releases/OpenLayers-2.9/doc/apidocs/files/OpenLayers-js.html. Ultimo accesso il 29/06/2010

Apache JMeter. http://jakarta.apache.org/jmeter/. Ultimo accesso il 27/06/2010

Apache JMeter: User's Manual. http://jakarta.apache.org/jmeter/usermanual/index.html. Ultimo accesso il 27/06/2010

# 9 Appendice: Codice Sorgente

Nel presente Appendice, verrà elencato il codice sorgente più rilevante sviluppato per il presente eleborato.

## *9.1 Strumenti di gestione delle osservazioni*

### 9.1.1 registerSensor.py

```
# import system modules
import sys
import string, httplib

# import defined modules
import configSOS
sys.path.append(sys.path[0]+"/../libs-python/")
import common_tools as tools
import SOS


def registerSensor_XML(sensorID):

    sensor = configSOS.sensors[sensorID]
    station = configSOS.foi[sensor['foi']]
    phenomenon = configSOS.phenomenoms[sensor['phenom']]
    procedure = sensor['phenom']
+'_'+string.upper(tools.replaceAccents(string.replace(station['name'], ' ',''')))
    feature_of_interest =
string.upper(tools.replaceAccents(string.replace(station['name'],' ', '')));
    dimension = '2'

    ## HEADER  ##
    xml = """<?xml version="1.0" encoding="UTF-8"?>
                <sos:RegisterSensor
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="http://schemas.opengis.net/sos/1.0.0/sosAll.xsd"
                    xmlns:sos="http://www.opengis.net/sos/1.0"
                    xmlns:gml="http://www.opengis.net/gml/3.2"
                    xmlns:ogc="http://www.opengis.net/ogc"
                    xmlns:om="http://www.opengis.net/om/1.0"
                    xmlns:xlink="http://www.w3.org/1999/xlink"
                    xmlns:swe="http://www.opengis.net/swe/0.0"
                    xmlns:sa="http://www.opengis.net/sampling/0.0"
```

```
                    service="SOS">"""

    ## SensorDescription ##
    xml += """<SensorDescription>
                <System>
                    <gml:description>%s</gml:description>""" %
(configSOS.sos['description']) + """
                    <gml:name>%s</gml:name>""" % (configSOS.sos['name']) + """
                    <keywords>
                        <KeywordList />
                    </keywords>
                    <identification>
                        <IdentifierList />
                    </identification>
                    <classification>
                        <ClassifierList>
                            <classifier name="Intended Application1">
                                <Term definition="urn:ogc:def:classifier:OGC:application">
                                    <value>%s</value>""" % (phenomenon['name']) + """
                                </Term>
                            </classifier>
                        </ClassifierList>
                    </classification>
                    <components>
                        <ComponentList>
                            <component>
                                <Component>
                                    <identification>
                                           <IdentifierList />
                                     </identification>
                                    <!-- INPUTS -->
                                    <inputs>
                                        <InputList>
                                            <input name="%s">""" % (sensor['phenom']) +
"""
                                                <swe:ObservableProperty
definition="urn:ogc:def:property:arpa::%s"/>""" % (sensor['phenom']) + """
                                            </input>
                                       </InputList>
                                    </inputs>
                                    <!-- OUTPUTS -->
                                    <outputs>
                                        <OutputList>
                                            <output name="%s">""" % (sensor['phenom']) +
"""
                                                <swe:Quantity
definition="urn:ogc:def:property:arpa::%s">""" % (sensor['phenom']) + """
                                                    <swe:uom code="%s"/>""" %
(phenomenon['units']) + """
                                                </swe:Quantity>
                                            </output>
                                       </OutputList>
                                    </outputs>
                                    <!-- METHOD -->
                                    <method
xlink:href="urn:ogc:def:process:1.0:detector"/>
                                </Component>
                            </component>
                        </ComponentList>
                    </components>
                </System>
             </SensorDescription>
            """
    ##  OBSERVATIONS ##
    xml += """<ObservationTemplate>
                <om:Observation>
                    <om:procedure xlink:href="urn:ogc:object:procedure:arpa::%s"/>""" %
(procedure) + """
                    <sa:samplingTime>
                        <gml:TimePeriod>
                            <gml:TimeLength>
```

```
                                    <gml:timeInterval unit="%s">1</gml:timeInterval>""" %
(sensor['intervalUnit']) + """
                                </gml:TimeLength>
                            </gml:TimePeriod>
                        </sa:samplingTime>
                        <om:observedProperty>
                            <swe:CompositPhenomenon dimension="2">
                                <swe:component
xlink:href="urn:ogc:def:parameter:arpa::time:iso8601"/>
                                <swe:component>
                                    <gml:name>%s</gml:name>""" % (sensor['phenom']) + """
                                    <gml:description>%s</gml:description>""" %
(phenomenon['name']) + """
                                </swe:component>
                            </swe:CompositPhenomenon>
                        </om:observedProperty>
                        <om:featureOfInterest
xlink:href="urn:ogc:object:feature:arpa::station:%s">""" % (feature_of_interest) + """
                            <gml:Point srsName="EPSG:%s">""" % (configSOS.sos['epsg']) +
"""
                                <gml:coordinates>%s,%s</gml:coordinates>""" %
(station['lat'], station['lon']) + """
                            </gml:Point>
                        </om:featureOfInterest>
                        <om:result>
                            <swe:DataArray>
                                <swe:field name="Time">
                                    <swe:Time
definition="urn:ogc:def:parameter:arpa::time:iso8601"/>
                                </swe:field>
                                <swe:field name="%s">""" % (sensor['phenom']) + """
                                    <swe:Quantity definition="urn:ogc:def:property:arpa::
%s">""" % (sensor['phenom']) + """
                                        <swe:uom code="%s"/>""" % (phenomenon['units']) +
"""
                                    </swe:Quantity>
                                </swe:field>
                            </swe:DataArray>
                        </om:result>
                    </om:Observation>
                </ObservationTemplate>"""

    xml += "</sos:RegisterSensor>"

    return xml


######## for each SENSOR send RegisterSensor request ########
for sensorID in configSOS.sensors.iterkeys():
    xml = registerSensor_XML(sensorID)
    SOS.sendRequest('registerSensor',xml)
```

## 9.1.2 insertObservation.py

```
# import system modules
import sys, getopt, os
import string, time

# import defined modules
sys.path.append(sys.path[0]+"/../libs-python/")
import configSOS
import istSOS_database as istSOS
import common_tools as tools
import SOS

#### GET ARGUMENTS ####
csvPath=''
dataType='singleData'
```

```
forceInsert='false'
args = sys.argv[1:]
optlist, args = getopt.getopt(args, '', ['csvPath=', 'dataType=', 'forceInsert='])
for opt, arg in optlist:
    if(opt=="--csvPath"):
        csvPath = arg
    elif(opt=="--dataType"):
        if(arg=="arrayData" or arg=='singleData'):
            dataType = arg
        else:
            dataType = ''
    elif(opt=="--forceInsert"):
        if(arg=="true" or arg=='false'):
            forceInsert = arg
        else:
            forceInsert = ''

# args Error
if(csvPath=="" or dataType=="" or forceInsert==""):
        errorMSG = "Argument ERROR\n"
        errorMSG += "  --csvPath (mandatory): csv file(s) location (relative path)\n"
        errorMSG += "  --dataType (optional):  'singleData' --> one insertObservation for
each observation\n"
        errorMSG += "                          'arrayData'  --> one insertObservation for
each file\n"
        errorMSG += "                         (default is 'singleData')\n"
        errorMSG += "  --forceInsert (optional):  force deletion and replacement of
existing observation (boolean)\n"
        errorMSG += "                            (default is false)\n"
        sys.exit(errorMSG)

def insertObservation_XML(sensorID, values, minEventTime, maxEventTime, forceInsert):

    sensor = configSOS.sensors[sensorID]
    station = configSOS.foi[sensor['foi']]
    phenomenon = configSOS.phenomenoms[sensor['phenom']]
    procedure = sensor['phenom']
+'_'+string.upper(tools.replaceAccents(string.replace(station['name'], ' ','')))
    feature_of_interest =
string.upper(tools.replaceAccents(string.replace(station['name'],' ', '')));
    assegnedID = istSOS.getAssignedID(procedure)

    ## HEADER  ##
    xml = """<?xml version="1.0" encoding="UTF-8"?>
                        <sos:InsertObservation
                           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                           xsi:schemaLocation="http://schemas.opengis.net/sos/1.0.0/sosAll
.xsd"
                           xmlns:sos="http://www.opengis.net/sos/1.0"
                           xmlns:xlink="http://www.w3.org/1999/xlink"
                           xmlns:sa="http://www.opengis.net/sampling/1.0"
                           xmlns:swe="http://www.opengis.net/swe/1.0.1"
                           xmlns:gml="http://www.opengis.net/gml/3.2"
                           xmlns:ogc="http://www.opengis.net/ogc"
                           xmlns:om="http://www.opengis.net/om/1.0" service="SOS"
version="1.0.0" >
        """

    ## forceInsert ##
    if(forceInsert=='true'):
        xml += "<forceInsert>true</forceInsert>"

    ## AssignedSensorId ##
    xml += "<AssignedSensorId>urn:ogc:object:sensor:arpa::%s</AssignedSensorId>" %
(assegnedID)

    ##  OBSERVATIONS ##
    xml += """<ObservationTemplate>
                <om:Observation>
                    <om:procedure xlink:href="urn:ogc:object:procedure:arpa::%s"/>""" %
(procedure) + """
```

```
                        <sa:samplingTime>
                            <gml:TimePeriod>
                                <gml:beginPosition>%s</gml:beginPosition>""" %
(minEventTime) + """
                                <gml:endPosition>%s</gml:endPosition>""" % (maxEventTime) +
"""
                            </gml:TimePeriod>
                        </sa:samplingTime>
                        <om:observedProperty>
                            <swe:CompositPhenomenon dimension="2">
                                <swe:component
xlink:href="urn:ogc:def:parameter:arpa::time:iso8601"/>
                                <swe:component>
                                    <gml:name>%s</gml:name>""" % (sensor['phenom']) + """
                                    <gml:description>%s</gml:description>""" %
(phenomenon['name']) + """
                                </swe:component>
                            </swe:CompositPhenomenon>
                        </om:observedProperty>
                        <om:featureOfInterest
xlink:href="urn:ogc:object:feature:arpa::station:%s">""" % (feature_of_interest) + """
                            <gml:Point srsName="EPSG:%s">""" % (configSOS.sos['epsg']) +
"""
                                <gml:coordinates>%s,%s</gml:coordinates>""" %
(station['lat'], station['lon']) + """
                            </gml:Point>
                        </om:featureOfInterest>
                        <om:result>
                            <swe:DataArray>
                                <swe:DataRecord
definition="http://mmiws.org/ont/x/timeSeries">
                                    <swe:field name="Time">
                                        <swe:Time
definition="urn:ogc:def:parameter:arpa::time:iso8601"/>
                                    </swe:field>
                                    <swe:field name="%s">""" % (sensor['phenom']) + """
                                        <swe:Quantity
definition="urn:ogc:def:property:arpa::%s">""" % (sensor['phenom']) + """
                                            <swe:uom code="%s"/>""" % (phenomenon['units'])
+ """
                                        </swe:Quantity>
                                    </swe:field>
                                </swe:DataRecord>
                                <swe:encoding>
                                    <swe:TextBlock tokenSeparator="," blockSeparator="@"
decimalSeparator="."/>
                                </swe:encoding>
                                <swe:values>
                                    %s """ % (values) + """
                                </swe:values>
                            </swe:DataArray>
                        </om:result>
                    </om:Observation>
                """

    xml += "</sos:InsertObservation>"

    return xml


#### read CSV files ####
dirList=os.listdir(csvPath)
for CSVfileName in dirList:

    #### read lines (observation) ####
    file = open(csvPath+CSVfileName)
    lines = file.readlines()

    # get sensor ID
    sensorID =  string.split(lines[0],',')[0]
    values = ''
```

```
    num = len(lines)
    i=0
    for line in lines:
        line = string.replace(line, "\n","")
        line = string.replace(line, "\r","")

        #### extract fields from csv
        observationField = string.split(line,',')
        eventTime = string.replace(observationField[1],' ', 'T')+'+01:00'
        measure = string.replace(observationField[2],',', '.')

        values += eventTime+','+measure+'@'

        if(i==0):
            minEventTime = eventTime
        elif(i==(num-1)):
            maxEventTime = eventTime
        i+=1

        if(dataType=='singleData'):
            values= eventTime+','+measure
            xml=insertObservation_XML(sensorID, values, eventTime, eventTime, forceInsert)
            print xml

    if(dataType=='arrayData'):
        values = string.rstrip(values, '@')
        xml = insertObservation_XML(sensorID, values, minEventTime, maxEventTime,
forceInsert)
        SOS.sendRequest('insertObservation',xml)
```

## 9.2 Strumenti di gestione delle medie

### 9.2.1 populateAverage.py

```
# import system modules
import sys, getopt
import string, calendar

# import defined modules
import istSOS_database as istSOS

# get args
observedProperty=foi=start=end=''
args = sys.argv[1:]
optlist, args = getopt.getopt(args, '', ['type=', 'observedProperty=', 'foi=',
'start=','end='])
for opt, arg in optlist:
    if(opt=="--type"):
        todo = arg
    elif(opt=="--observedProperty"):
        observedProperty = arg
    elif(opt=="--foi"):
        foi = arg
    elif(opt=="--start"):
        start = arg.split('-')
    elif(opt=="--end"):
        end = arg.split('-')


if(observedProperty==''):
    observedProperties = istSOS.getObservedProperties()
```

```
else:
    observedProperties = [observedProperty]

if(foi==''):
    fois = istSOS.getFois()
else:
    fois = [foi]



def setEventLimitations(rangeFromDB, startFromArgs, endFromArgs):
    range = {}

    if(startFromArgs!=''):
        range['start_year'] = startFromArgs[0]
        range['start_month'] = startFromArgs[1]
    else:
        range['start_year'] = rangeFromDB['start_year']
        range['start_month'] = rangeFromDB['start_month']

    if(endFromArgs!=''):
        range['end_year'] = endFromArgs[0]
        range['end_month'] = endFromArgs[1]
    else:
        range['end_year'] = rangeFromDB['end_year']
        range['end_month'] = rangeFromDB['end_month']

    return range

## foreach observedProperty
for observedProperty in observedProperties:

    ## foreach foi
    for foi in fois:
        if(istSOS.procedureExists(observedProperty, foi)==True):

            procedure = istSOS.getProcedure(observedProperty, foi)
            eventRange = setEventLimitations(istSOS.getProcedureRange(procedure), start,
end)

            ## remove selected interval from DB
            sql = 'DELETE FROM "public"."MonthlyAverage" WHERE '
            sql += '"observedProperty"=\'%s\'' % (observedProperty);
            sql += ' AND "foi"=\'%s\'' % (foi);
            sql += ' AND "eventTime" BETWEEN \'%s-%s-01T00:00:00+01:00\'' %
(eventRange['start_year'], eventRange['start_month'])
            sql += ' AND \'%s-%s-01T00:00:00+01:00\';' % (eventRange['end_year'],
eventRange['end_month'])
            print sql

            if(eventRange['start_year']==eventRange['end_year']):
                availableYears = [int(eventRange['start_year'])]
            else:
                availableYears = range(int(eventRange['start_year']),
int(eventRange['end_year'])+1)

            ## foreach year
            for y in availableYears:

                if(eventRange['start_year']==eventRange['end_year']):
                    availableMonths = range(int(eventRange['start_month']),
int(eventRange['end_month'])+1)
                elif(y==int(eventRange['start_year'])):
                    availableMonths = range(int(eventRange['start_month']), 13)
                elif(y==int(eventRange['end_year'])):
                    availableMonths = range(1, int(eventRange['end_month'])+1)
                else:
                    availableMonths = range(1,13)

                ## foreach month
                for m in availableMonths:
```

```
                    eventTime = '%s-%s-01T00:00:00+01:00' %(y, '%02d' % m)
                    if(istSOS.procedureExists(observedProperty, foi, eventTime)==True):
                       measure = istSOS.getMonthlyAverage(observedProperty, foi, y, m)
                       if(measure!=None):
                           sql = 'INSERT INTO "public"."MonthlyAverage"
("observedProperty","foi","measure","eventTime") VALUES '
                           sql += "('%s','%s','%s','%s');" %(observedProperty, foi,
measure, eventTime)
                           pgdb = istSOS.istSOS_connection()
                           result = pgdb.execute(sql)
```

## 9.3 Processi WPS

### 9.3.1 interpolation.py

```
# import system modules
import sys, os
import time, types, string
import json

# import defined modules
sys.path.append(sys.path[0]+"/../libs-python/")
import istSOS_database as istSOS
import common_tools as tools
import grass
from pywps.Process.Process import WPSProcess


class Process(WPSProcess):

    def __init__(self):

        WPSProcess.__init__(self,
             identifier = "interpolation", # must be same, as filename
             title="Interpolazione Sensori (GRASS v.surf.idw)",
             version = "0.1",
             grassLocation = "newLocation")

        # inputs
        self.observedProperty = self.addLiteralInput(identifier="observedProperty",
                                                     type = types.StringType,
                                                     title="Proprieta Osservata")
        self.eventTime = self.addLiteralInput(identifier="eventTime",
                                              type = types.StringType,
                                              title="Istante")
        self.tempUrl = self.addLiteralInput(identifier="tempUrl",
                                              type = types.StringType,
                                              title="Image temporary URL")
        # outputs
        self.imageName = self.addLiteralOutput(identifier="imageName",
                                               type = types.StringType,
                                               title="Output Image filename")
        self.executionTime = self.addLiteralOutput(identifier="executionTime",
                                               type = types.StringType,
                                               title="script execution Time")
        self.extent = self.addLiteralOutput(identifier="extent",
                                               type = types.StringType,
                                               title="Output bounding box")


    def execute(self):
```

```
        startTime = time.time()
        resolution = 50
        tempPath = '/var/www/pywps/temp/'
        tempUrl = '%spywps/temp/' % (self.tempUrl.getValue())


        # clean temp directory
        tools.cleanDir(tempPath)

        # set filenames
        stringTime = time.strftime("%H-%M-%S")
        imageFile = 'interpolation%s.png' % (stringTime)
        pointsFile = '%spoints%s.txt' % (tempPath, stringTime)

        # get points from DB
        sensors = istSOS.getPoints(self.observedProperty.getValue(),
self.eventTime.getValue())
        os.system("echo '%s'  > %s" %(sensors, pointsFile))


        ###  STEP 1: generate vector layer (from points)  ###
        tableColumns = "foi varchar(40), x double precision, y double precision, measure
double precision"
        self.cmd(["v.in.ascii","input=%s" %
(pointsFile),"output=mask","x=2","y=3","cat=0","columns=%s" % (tableColumns)])

        ###  STEP 2: set region (from vector) ###
        self.cmd(["g.region","vect=mask", "res=%s" % resolution])
        gregion = self.cmd("g.region -g")
        gregion = grass.parse_gregion(gregion) # to JSON string
        gregion = json.loads(gregion)   # decode JSON string to python object
        self.cmd(["g.region",
                "w=%s" % str(int(gregion['w'])-1000),
                "s=%s" % str(int(gregion['s'])-1000),
                "e=%s" % str(int(gregion['e'])+1000),
                "n=%s" % str(int(gregion['n'])+1000),
                "res=%s" % resolution])
        extent = self.cmd("g.region -g")

        ###  STEP 3: generate interpolation (from vector layer)  ###
        self.cmd("v.surf.idw input=mask output=grid column=measure")

        ###  STEP 4: generate image  ###
        self.cmd("r.colors map=grid color=gyr")
        self.cmd("r.out.png input=grid output=%s" % (tempPath+imageFile))

        # OUTPUT
        endTime = time.time()
        executionTime = round(endTime-startTime, 3)

        self.executionTime.setValue('executed in %s seconds.' % (executionTime))
        self.imageName.setValue(tempUrl+imageFile)
        self.extent.setValue(grass.parse_gregion(extent))
        return
```

## 9.3.2 montlhyAverages.py

```
# import system modules
import sys
import time, types, json

# import defined modules
sys.path.append(sys.path[0]+"/../libs-python/")
import istSOS_database as istSOS
from pywps.Process.Process import WPSProcess


class Process(WPSProcess):
```

```
def __init__(self):

     WPSProcess.__init__(self,
          identifier = "monthlyAverages", # must be same, as filename
          title="Medie mensili",
          version = "0.1")

     # inputs
     self.observedProperty = self.addLiteralInput(identifier="observedProperty",
                                                  type = types.StringType,
                                                  title="Proprieta Osservata")
     self.year = self.addLiteralInput(identifier="year",
                                      type = types.StringType,
                                      title="Anno")
     # outputs
     self.means = self.addLiteralOutput(identifier="means",
                                        type = types.StringType,
                                        title="Means")
     self.executionTime = self.addLiteralOutput(identifier="executionTime",
                                        type = types.StringType,
                                        title="script execution Time")

def execute(self):

    startTime = time.time()
    locations = istSOS.getFoiLocations()
    units = istSOS.getUnits()
    output = {}

    ################  extract means values FROM mean's tables  ################
    for month in range(1,13):

        pgdb = istSOS.istSOS_connection()
        sql = 'SELECT "measure", "foi" FROM "public"."MonthlyAverage" WHERE '
        sql += '"observedProperty"=\'%s\' ' % (self.observedProperty.getValue())
        sql += 'AND date_part(\'year\', "eventTime")=\'%s\' ' %(self.year.getValue())
        sql += 'AND date_part(\'month\', "eventTime")=\'%s\' ' %(month)
        sql += 'ORDER by "eventTime";'
        result = pgdb.execute(sql)

        points = []
        for record in result:
            mean = {}
            mean['foi'] = record[1]
            mean['location'] = locations[mean['foi']]
            mean['mean'] = float(record[0])
            mean['observedProperty'] = self.observedProperty.getValue()
            mean['period'] = '%s-%s' % (month, self.year.getValue())
            mean['unit'] = units[mean['observedProperty']]

            points.append(mean)

        output[month] = points

    # OUTPUT
    endTime = time.time()
    executionTime = round(endTime-startTime, 3)
    self.executionTime.setValue('executed in %s seconds.' % (executionTime))

    self.means.setValue(json.dumps(output))
    return
```

## 9.4 Classi Openlayers

### 9.4.1 client.istSOS.js

```
/**
 *
 * Class: OpenLayers.istSOS
 *
 */
       OpenLayers.istSOS = OpenLayers.Class({

                istSOSurl: null,
                format: null,
                executionTime: null,
                layerSOS: null,

            /**
             * Constructor: OpenLayers.istSOS
             *
             * Parameters:
             * istSOSurl - {String}
             */
                initialize: function(istSOSurl) {

                        var paramString = OpenLayers.Util.getParameterString({
                                                'service': 'SOS',
                                                'version': '1.0.0',
                                                'request': 'getObservation',
                                                'responseFormat':
'text/xml;subtype=\'sensorML/1.0.0\''});
                        this.istSOSurl = istSOSurl;
                        this.istSOSurl = urlAppend(this.istSOSurl, paramString);

                        this.format = new OpenLayers.Format.istSOSGetObservation();

            },

            /**
             * Methode: execute
             *
             * Parameters:
             * url - {String}
             * callback - {Function}
             */
                execute: function(url, callback){

                        OpenLayers.Request.GET({url: url,
                                                scope: this,
                                                async: false,
                                                callback: callback});
                },

            /**
             * Methode: drawSensors
             *
             * Parameters:
             * params - {Object}
             */
                drawSensors: function(observedProperty, eventTime) {

                        // ## inputs ##
                        var currentUrl = this.istSOSurl+'&offering=temporary';
                        currentUrl += '&observedProperty='+observedProperty;
                        currentUrl += '&eventTime='+eventTime;

                        // ## callback ##
                        var callback = function(ajaxResponse){
```

```
                                var response = this.format.read(ajaxResponse.responseXML
|| ajaxResponse.responseText);

                                // ## outputs ##
                                var fois = response.observations;
                                var numFois = fois.length;

                                if(numFois==0){
                                        alert("La banca dati non contiene osservazioni \n
che rispondono ai parametri selezionati");
                                        return false;
                                } else
                                        this.layerSOS = new
OpenLayers.Layer.Features.istSOS('istSOS', fois);
                                        return false;
                        }

                        // ## execute ##
                        this.execute(currentUrl, callback);

                },

                CLASS_NAME: "OpenLayers.istSOS"
        });
```

## 9.4.2 format.istSOS.GetObservation.js

```
/**
 * @requires OpenLayers/Format/XML.js
 */

/**
 * Class: OpenLayers.Format.istSOSGetObservation
 * Read and write SOS version 1.0.0
 *
 * Inherits from:
 *  - <OpenLayers.Format.XML>
 */
OpenLayers.Format.istSOSGetObservation = OpenLayers.Class(OpenLayers.Format.XML, {

      /**
     * Constant: VERSION
     * {String} 1.0.0
     */
    VERSION: "1.0.0",

    /**
     * Property: namespaces
     * {Object} Mapping of namespace aliases to namespace URIs.
     */
    namespaces: {
        om: "http://www.opengis.net/om/1.0",
        gml: "http://www.opengis.net/gml",
        xsi: "http://www.w3.org/2001/XMLSchema-instance",
                xlink: "http://www.w3.org/1999/xlink",

        sa: "http://www.opengis.net/sampling/1.0",
                swe: "http://www.opengis.net/swe/1.0.1"
    },

      /**
     * Property: schemaLocation
     * {String} Schema location
     */
```

```
   // schemaLocation: "http://www.opengis.net/sos/1.0
http://schemas.opengis.net/sos/1.0.0/sosAll.xsd",

       /**
     * Property: defaultPrefix
     */
   // defaultPrefix: "sos",

       /**
     * Property: regExes
     * Compiled regular expressions for manipulating strings.
     */
    //regExes: {
    //    trimSpace: (/^\s*|\s*$/g),
    //    removeSpace: (/\s*/g),
    //    splitSpace: (/\s+/),
    //    trimComma: (/\s*,\s*/g)
    //},


       /**
     * Constructor: OpenLayers.Format.istSOSGetObservation
     *
     * Parameters:
     * options - {Object} An optional object whose properties will be set on
     *     this instance.
     */
    initialize: function(options) {
        OpenLayers.Format.XML.prototype.initialize.apply(this, [options]);
    },

        /**
     * Method: read
     *
     * Parameters:
     * data - {String} or {DOMElement} data to read/parse.
     *
     * Returns:
     */
    read: function(data) {
               if(typeof data == "string") {
            data = OpenLayers.Format.XML.prototype.read.apply(this, [data]);
        }
        if(data && data.nodeType == 9) {
            data = data.documentElement;
        }
        var info = {observations: []};
        this.readNode(data, info);
        return info;
       },

       /**
     * Property: readers
     * Contains public functions, grouped by namespace prefix, that will
     *     be applied when a namespaced node is found matching the function
     *     name.  The function will be applied in the scope of this parser
     *     with two arguments: the node being read and a context object passed
     *     from the parent.
     */
    readers: {
                "om": {
            "ObservationCollection": function(node, obj) {
                this.readChildNodes(node, obj);
            },
            "member": function(node, obj) {
                this.readChildNodes(node, obj);
            },
            "Observation": function(node, obj) {
                var observation = {};
                obj.observations.push(observation);
                this.readChildNodes(node, observation);
```

```
            },
                    "procedure": function(node, obj) {
                obj.procedure = this.getAttributeNS(node, this.namespaces.xlink, "href");
                this.readChildNodes(node, obj);
            },
                    "featureOfInterest": function(node, obj) {
                obj.featureOfInterest = {};
                        var foi = obj.featureOfInterest;
                        foi.Name = this.getAttributeNS(node,
this.namespaces.xlink, "href");
                this.readChildNodes(node, obj.featureOfInterest);
            },
                    "observedProperty": function(node, obj) {
                obj.observedProperty = {};
                this.readChildNodes(node, obj.observedProperty);
            },
                    "result": function(node, obj) {
                obj.result = {};
                this.readChildNodes(node, obj.result);
            }
        },
            "swe": {
                    "CompositPhenomenon": function(node, obj) {
                obj.id = node.getAttribute("id");
                obj.dimension = node.getAttribute("dimension");
                obj.components = [];
                this.readChildNodes(node, obj.components);
            },
                    "component": function(node, obj) {
                obj.push(this.getAttributeNS(node, this.namespaces.xlink, "href"));
            },
                    "DataArray": function(node, obj) {
                this.readChildNodes(node, obj);
            },
                    "elementType": function(node, obj) {
                this.readChildNodes(node, obj);
            },
                    "DataRecord": function(node, obj) {
                //obj.fields = [];
                        obj.fields = [];
                this.readChildNodes(node, obj);
            },
                    "field": function(node, obj) {
                        var field = {};
                obj.fields.push(field);
                this.readChildNodes(node, field);
            },
            "Quantity": function(node, obj) {
                        obj.name = node.parentNode.getAttribute("name");
                        this.readChildNodes(node, obj);
                    },
                    "uom": function(node, obj) {
                obj.uom = node.getAttribute("code");
            },
            "encoding": function(node, obj) {
                this.readChildNodes(node, obj);
            },
            "TextBlock": function(node, obj) {
                obj.decimalSeparator = node.getAttribute("decimalSeparator");
                obj.tokenSeparator = node.getAttribute("tokenSeparator");
                obj.blockSeparator = node.getAttribute("blockSeparator");
                this.readChildNodes(node, obj);
            },
                    "values": function(node, obj) {
                var text = this.getChildValue(node);
                var records = text.split(obj.blockSeparator);
                var values = [];
                        for (var i=0; i<records.length; i++) {
                    var record = records[i];
                    var row = record.split(obj.tokenSeparator);
                    var result = [];
```

```
                    for (var j=0, lenj=row.length; j<lenj; j++) {
                        result.push(row[j]);
                    }
                    values.push(result);
                }
                obj.values = values;
            }
                },
         "sa": {
                    "samplingTime": function(node, obj) {
                obj.samplingTime = {};
                this.readChildNodes(node, obj.samplingTime);
            },
                },
                "gml": OpenLayers.Util.applyDefaults({
                        "TimePeriod": function(node, obj) {
                obj.timePeriod = {};
                this.readChildNodes(node, obj.timePeriod);
            },
            "beginPosition": function(node, obj) {
                obj.beginPosition = this.getChildValue(node);
            },
            "endPosition": function(node, obj) {
                obj.endPosition = this.getChildValue(node);
            },
                    "TimeLength": function(node, obj){
                this.readChildNodes(node, obj);
                    },
                    "duration": function(node, obj) {
                obj.duration = this.getChildValue(node);
            },
            "timeInterval": function(node, obj) {
                obj.timeInterval = this.getChildValue(node);
                            obj.timeIntervalUnit =  node.getAttribute("unit");
            },
                    "Point": function(node, obj) {
                obj.point = {};
                this.readChildNodes(node, obj.point);
            },
                    "coordinates": function(node, obj) {
                var coordinates = this.getChildValue(node).split(',');
                            obj.x = coordinates[0];
                            obj.y = coordinates[1];
                            obj.srsName = node.parentNode.getAttribute("srsName");
            },
                }, OpenLayers.Format.GML.v3.prototype.readers.gml)
        },


        CLASS_NAME: "OpenLayers.Format.istSOSGetObservation"

});
```

## 9.4.3 layer.Features.istSOS.js

```
/**
 * @requires OpenLayers/Layer/Features.js
 *
 * Class: OpenLayers.Layer.Features.istSOS
 *
 * Inherits from:
 *
 *  - <OpenLayers.Layer.Features>
 */
      OpenLayers.Layer.Features.istSOS = OpenLayers.Class( OpenLayers.Layer.Features, {

            /**
         * Method: exist
```

```
                *
                * Parameters:
                * foi - {JSON}
                */
                    exist: function(foi){
                            if(foi.result.values[0][1]!=undefined)
                                    return true;
                            else
                                    return false
                    },

                    /**
                * Method: markerIcon
                *
                * Parameters:
                * foi - {JSON}
                */
                    markerIcon: function(foi){

                            var observedProperty = foi.result.fields[1].name;
                            var measureValue = foi.result.values[0][1];

                            var iconFile = setMarkerColor(measureValue, observedProperty);
                            var size = new OpenLayers.Size(15, 15);
                            var icon = new OpenLayers.Icon(this.markerUrl + iconFile, size);
                            return icon;
                    },

                    /**
                * Method: markerLocation
                *
                * Parameters:
                * foi - {JSON}
                */
                    markerLocation: function(foi){

                            var point = foi.featureOfInterest.point;
                        var location = new OpenLayers.LonLat(point.x, point.y);
                            return location;
                    },

                    /**
                * Method: popupContent
                *
                * Parameters:
                * foi - {JSON}
                */
                    popupContent: function(foi){

                            var foiName = getNameFromURN(foi.featureOfInterest.Name);
                            var observedProperty = foi.result.fields[1].name;
                            var observedPropertyUnit = foi.result.fields[1].uom;
                            var measureDate = foi.result.values[0][0];
                            var measureValue = foi.result.values[0][1];

                            var data = {};
                            data.popupContentHTML = '<b>' + foiName + '</b> (' +
observedProperty + ')<br />';
                            data.popupContentHTML += measureValue + ' ' + observedPropertyUnit
+ ' @ ' + measureDate;
                            return data;
                    },


        CLASS_NAME: "OpenLayers.Layer.Features.istSOS"
        });
```

### 9.4.4 client.pyWPS.js

```
/**
 *
 * Class: OpenLayers.pyWPS
 *
 */
      OpenLayers.pyWPS = OpenLayers.Class({

              pyWPSurl: null,
              format: null,
              executionTime: null,
              layerWPS: null,

            /**
             * Constructor: OpenLayers.pyWPS
             *
             * Parameters:
             * pyWPSurl - {String}
             */
                initialize: function(pyWPSurl) {

                        var paramString = OpenLayers.Util.getParameterString({
                                                'service': 'wps',
                                                'version': '1.0.0',
                                                'request': 'execute'});
                        this.pyWPSurl = pyWPSurl;
                        this.pyWPSurl = urlAppend(this.pyWPSurl, paramString);

                        this.format = new OpenLayers.Format.WPS();

            },

            /**
             * Methode: execute
             *
             * Parameters:
             * url - {String}
             * callback - {Function}
             */
                execute: function(url, callback){

                        OpenLayers.Request.GET({url: url,
                                                scope: this,
                                                async: false,
                                                callback: callback});
                },


                /*  ################################################
                    ###########  Process: interpolation  ###########
                    ################################################  */
                temp_url: null,
                /**
                 * Methode: interpolation
                 *
                 * observedProperty - {String}
                 * eventTime - {String}
                 * bbox - {OpenLayers.Bounds}
                 */
                interpolation: function(observedProperty, eventTime) {

                        // ## inputs ##
                        var inputs = 'observedProperty='+observedProperty+';';
                        inputs += 'eventTime='+eventTime+';';
                        inputs += 'tempUrl='+this.temp_url+';';
                        var currentUrl =
this.pyWPSurl+'&identifier=interpolation&datainputs='+inputs;

                        // ## callback ##
```

```
                        var callback = function(ajaxResponse) {

                                var response = this.format.read(ajaxResponse.responseXML
|| ajaxResponse.responseText);

                                // ## outputs ##
                                this.executionTime = response.processoutputs[0].Value;
                                var fileImageURL = response.processoutputs[1].Value;
                                var extent = response.processoutputs[2].Value;
                                var jsonParser = new OpenLayers.Format.JSON();
                                var extentJSON = jsonParser.read(extent);
                                var bbox = new OpenLayers.Bounds(extentJSON['w'],
                                                                 extentJSON['s'],
                                                                 extentJSON['e'],
                                                                 extentJSON['n']);

                                var img = new Image();
                                img.src = fileImageURL;
                                var size = new OpenLayers.Size(img.width, img.height)
                                this.layerWPS = new OpenLayers.Layer.Image('WPSLayer',
fileImageURL, bbox, size, {             isBaseLayer: false,
                                        resolutions: mapClient.mapResolutions      });

                                this.layerWPS.setOpacity(0.7);



                                return false;
                                }

                        // ## execute ##
                        this.execute(currentUrl, callback);
                        },


                        /*  ##############################################
                            ##########  Process: monthlyAverages  ##########
                            ##############################################  */

                        /**
                         * Methode: monthlyAverages
                         *
                         * Parameters:
                         * observedProperty - {String}
                         * year - {String}
                         */
                        monthlyAverages: function(observedProperty, year) {

                                // ## inputs ##
                                var inputs = 'observedProperty='+observedProperty+';year='+year;
                                var currentUrl =
this.pyWPSurl+'&identifier=monthlyAverages&datainputs='+inputs;

                                // ## callback ##
                                var callback = function(ajaxResponse){

                                        var response = this.format.read(ajaxResponse.responseXML
|| ajaxResponse.responseText);

                                        // ## outputs ##
                                        this.executionTime = response.processoutputs[0].Value;
                                        var data =
jQuery.parseJSON(response.processoutputs[1].Value);

                                        this.layersWPS = {}
                                        for(var month=1;month<13;month++){
                                                if(typeof(data[month])!="undefined") {
                                                        this.layersWPS[month] = new
OpenLayers.Layer.Features.pyWPS('pyWPS'+month, data[month]);
                                                }
                                        }
```

```
                return false;
            }

            // ## execute ##
            this.execute(currentUrl, callback);
        },

        CLASS_NAME: "OpenLayers.pyWPS"
    });
```

### 9.4.5 layer.Features.pyWPS.js

```
/**
 * @requires OpenLayers/Layer/Features.js
 *
 * Class: OpenLayers.Layer.Features.pyWPS
 *
 * Inherits from:
 *
 *  - <OpenLayers.Layer.Features>
 */
    OpenLayers.Layer.Features.pyWPS = OpenLayers.Class( OpenLayers.Layer.Features, {

        /**
     * Method: markerIcon
     *
     * Parameters:
     * foi - {JSON}
     */
        markerIcon: function(foi){

            var observedProperty = foi['observedProperty'];
            var measureValue = foi['mean'];

            var iconFile = setMarkerColor(measureValue, observedProperty);
            var size = new OpenLayers.Size(15, 15);
            var icon = new OpenLayers.Icon(this.markerUrl + iconFile, size);
            return icon;
        },

        /**
     * Method: markerLocation
     *
     * Parameters:
     * foi - {JSON}
     */
        markerLocation: function(foi){

            var location = new OpenLayers.LonLat(foi['location'][0],
foi['location'][1]);
            return location;
        },

        /**
     * Method: popupContent
     *
     * Parameters:
     * foi - {JSON}
     */
        popupContent: function(foi){

            var foiName = foi['foi'];
            var observedProperty = foi['observedProperty'];
            var measureValue = foi['mean'];
            var measureDate = foi['period'];
            var observedPropertyUnit = foi['unit'];
```

```
                        var data = {};
                        data.popupContentHTML = '<b>'+foiName+'</b>
('+observedProperty+')<br />';
                        data.popupContentHTML += measureValue+' '+observedPropertyUnit+' @
'+measureDate;
                        return data;
                },


        CLASS_NAME: "OpenLayers.Layer.Features.pyWPS"
        });
```

## 9.4.6 layer.Featurs.js

```
/**
 * @requires OpenLayers/Layer/Markers.js
 *
 * Class: OpenLayers.Layer.Features
 *
 * Inherits from:
 *
 *  - <OpenLayers.Layer.Markers>
 */
        OpenLayers.Layer.Features = OpenLayers.Class( OpenLayers.Layer.Markers, {

                isBaseLayer: false,
                features: [],
                markerUrl: null,

                /**
            * Constructor: OpenLayers.Layer.Features
            *
            * Parameters:
            * name - {String}
            * fois - {JSON}
            * options - {Object}
            */
                initialize: function(name, fois, options) {
                        OpenLayers.Layer.Markers.prototype.initialize.apply(this, [name,
options]);

                        // ##draw one feature (marker+popup) for each foi ##
                        for (var i in fois){
                                if(this.exist(fois[i])==true)
                                        this.createFeature(fois[i]);
                        }
                },

                /**
            * Method: exist
            *
            * Parameters:
            * foi - {JSON}
            */
                exist: function(foi){
                        // ## implementation in sub-classes (if necessary) ##
                        return true;
                },

                /**
            * Method: createFeature
            *
            * Parameters:
            * foi - {JSON}
            */
                createFeature: function(foi){

                        var data = this.popupContent(foi);
```

```
                var location = this.markerLocation(foi);

                // ## create marker ##
                if(typeof(markerUrl)!=undefined){
                        this.markerUrl = markerUrl;
                        var icon = this.markerIcon(foi);
                }
                else
                       var icon = null;
                var marker = new OpenLayers.Marker(location, icon);

                // ## create feature ##
                var feature = new OpenLayers.Feature(this, location, data);
                feature.popupClass = OpenLayers.Popup;
                feature.marker = marker;

                // ## add marker to Layer  ##
                this.addMarker(marker);

                // ## add feature to Layer  ##
                marker.events.register('click', feature, this.showPopup);
                this.features.push(feature);

        },

        /**
     * Method: markerIcon
     *
     * Parameters:
     * foi - {JSON}
     */
        markerIcon: function(foi){
                // ## implementation in sub-classes ##
        },

        /**
     * Method: markerLocation
     *
     * Parameters:
     * foi - {JSON}
     */
        markerLocation: function(foi){
                // ## implementation in sub-classes ##
        },

        /**
     * Method: popupContent
     *
     * Parameters:
     * foi - {JSON}
     */
        popupContent: function(foi){
                // ## implementation in sub-classes ##
        },

        /**
    * Method: showPopup
    *
    * Parameters:
    * evt - {Event}
    */
        showPopup: function(evt) {

                var sameMarkerClicked = (this == this.layer.selectedFeature);
        this.layer.selectedFeature = (!sameMarkerClicked) ? this : null;

                for(var i=0; i < this.layer.map.popups.length; i++) {
            this.layer.map.removePopup(this.layer.map.popups[i]);
        }

                if (!sameMarkerClicked) {
```

```
                        var popup = this.createPopup();
                        popup.closeOnMove = true;
                        popup.autoSize = true;
                        popup.backgroundColor = '#EEFFEE';
                        popup.border = '1px solid #228b22';
                        popup.opacity = 0.9;
                    OpenLayers.Event.observe(popup.div, "click",
                        OpenLayers.Function.bind(function() {
                            for(var i=0; i < this.layer.map.popups.length; i++) {
                                this.layer.map.removePopup(this.layer.map.popups[i]);
                            }
                        }, this)
                    );
                    this.layer.map.addPopup(popup);
                }
                OpenLayers.Event.stop(evt);
                 },


                 /**
            * Method: clearFeatures
            * Destroy all features in this layer.
            */
                 clearFeatures: function() {
                if(this.features != null) {
                    while(this.features.length > 0) {
                        var feature = this.features[0];
                        OpenLayers.Util.removeItem(this.features, feature);
                        feature.destroy();
                    }
                }
            },


                 CLASS_NAME: "OpenLayers.Layer.Features"
});
```