1. Premessa

Il rapido diffondersi delle tecnologie di networking ha stimolato, negli ultimi anni, una accresciuta esigenza di paradigmi adatti allo sviluppo di applicazioni distribuite. Tali paradigmi si fondano sul modello concettuale del Remote Procedure Calling (RPC) ed è stato più volte reimplementato in molte architetture: CORBA, COM/DCOM, RMI, etc.

Il modello RPC, alla base della maggior parte dei modelli client/server, prende le mosse dalla "semplice" idea di modellare la comunicazione come una chiamata, una "call" appunto, ad una routine che, potenzialmente, può essere locata su un sistema accessibile via rete.

Su questo paradigma, nel tempo, sono state costruite architetture distribuite che hanno ulteriormente raffinato il modello, modellando insiemi di routine come "oggetti", sulla base di architetture, tecnologie e paradigmi object oriented.

Le architetture che ne sono derivate hanno trovato largo uso nello sviluppo di applicazioni distribuite in uso all'interno di grandi organizzazioni, ma hanno mostrati altresì seri limiti quanto si è tentato di utilizzarle nella realizzazione di applicazioni in cui diviene necessario "scavalcare" la frontiera di una unità organizzativa.

L'affermarsi di sistemi dedicati alla protezione ed alla sicurezza della rete ha infatti creato "barriere" logiche che impediscono alle applicazioni di comunicare utilizzando protocolli e porte di comunicazione non standard. Inoltre le problematiche legate alla sicurezza hanno e stanno imponendo la standardizzazione di protocolli che garantiscano l'integrità e la sicurezza dei dati applicativi che vengono scambiati nelle interazioni.

Sempre di più viene avvertita l'esigenza di framework di sviluppo applicativo che permettano di:

a) utilizzare protocolli standard compatibili con le attuali infrastrutture tecnologiche;

b) ricomporre sistemi in nuove applicazioni;

c) utilizzare modelli che garantiscano elevati standard di sicurezza e di affidabilità.

2. Il ruolo dell'XML

L'Extensible Markup Language, abbreviato nell'acronimo XML è una grammatica BNF disegnata per:

1. essere facilmente utilizzabile su Internet;
2. essere il supporto di una grande varietà di applicazioni;
3. essere compatibile con lo standard SGML (Standard Generalized Markup Language), di cui è una sottogrammatica;
4. rendere semplice la stesura di programmi che processano documenti, stringhe, XML;
5. rendere inutile la nascita di "dialetti" XML;
6. permettere che i documenti XML siano facilmente leggibili da un essere umano;
7. semplificare il disegno di architetture fondate sull'XML;
8. rendere formale e coinciso il disegno di sistemi XML;
9. rendere facile la creazione di documenti XML;

L'XML definisce l'insieme delle stringhe appartenenti alla grammatica e permette la definizione di nuove sottogrammatiche, di nuovi linguaggi che possono essere facilmente utilizzati per molteplici usi:

a. descrizione di dati;
b. definizione di protocolli;
c. definizione del formato di documenti.

In generale una architettura, un sistema o una applicazione XML utilizzano uno "schema" per definire quali documenti appartengono al dominio della applicazione. In effetti definiscono una sottogrammatica, un sottolinguaggio, sul quale può essere implementato un "parser" in grado di eseguire azioni "semantiche" basate sulla struttura del documento.

Per un semplice esempio possiamo pensare a documenti che definiscano "Insiemi" di "Elementi" che siamo coppie di stringhe (Nome, Valore). Dall'insieme { (Nome1, Valore1), (Nome2, Valore2) } può facilmente essere costruito un documento XML di questa forma:

```
<?xml version="1.0"?>
<Insieme
        xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
        xsi:noNamespaceSchemaLocation='Insieme.xsd'>

        <Elemento>
                <Nome>Elementot1</Nome>
                <Valore>Valore1</Valore>
        </Elemento>

        <Elemento>
                <Nome>Elementot2</Nome>
                <Valore>Valore2</Valore>
        </Elemento>

</Insieme>
```

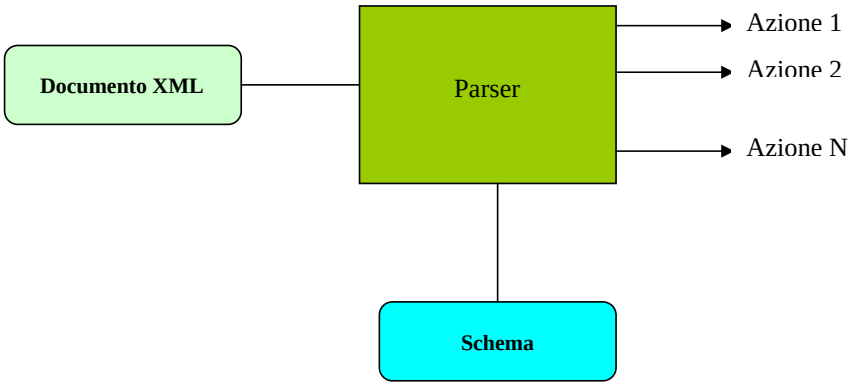
la cui grammatica è definita nello schema:

```
<?xml version="1.0" encoding="UTF-8"?>

  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="Insieme">
        <xsd:complexType>
            <xsd:sequence>
            <xsd:element ref="Elemento" minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="Elemento">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="Nome" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="Valore" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="Nome"   type="xsd:string"/>
    <xsd:element name="Valore" type="xsd:string"/>

  </xsd:schema>
```

che è a sua volta un documento XML.

L’XML può essere utilizzato facilmente per scambiare dati tra sistemi eterogenei: può infatti descrivere una qualsiasi collezione di dati ed essere quindi utilizzato per definire protocolli applicativi atti alla codifica ed allo scambio di messaggi tra applicazioni.

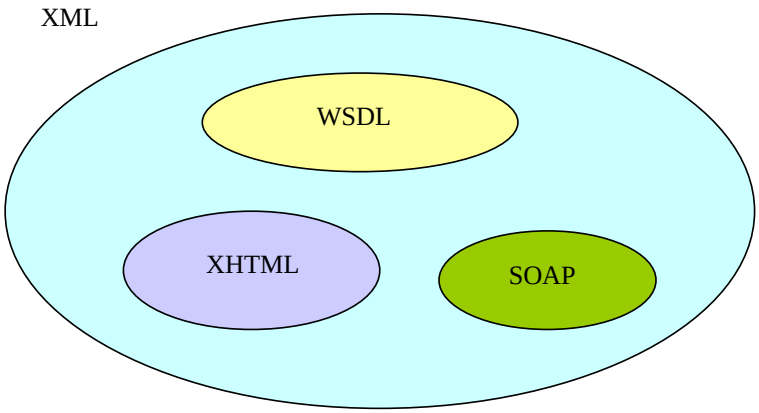
Nel modello RPC può quindi essere utilizzato per implementare i protocolli che si trovano al cuore della chiamata a procedure remote.



Il ruolo dell'XML è ben diverso dal quello dell'HTML: sebbene possa essere utilizzato direttamente da un browser, è stato disegnato per essere il "motore" di sistemi in grado di eseguire azioni "semantiche" fondate sulla struttura "formale" di un documento XML.

Ciò ovviamente non esclude l'uso dell'XML come formato di documenti fruibili direttamente dall'utente finale. L'XHTML è una sottogrammatica XML che formalizza in modo rigoroso la struttura sintatticamente *debole* dell'HTML.

Il *parser* XML più semplice che può essere costruito è rappresentato da un **validatore**, ovvero un parser in grado di stabilire se un documento, una stringa, XML appartiene o no ad una particolare sottogrammatica XML. In questo, il più semplice, caso una ed una sola azione semantica viene compiuta: il validatore restituisce un valore booleano (vero o falso) che determina se il documento XML appartiene alla classe di tutte le stringhe che appartengono ad una particolare grammatica. In questo contento le grammatiche XML possono essere interpretate come "regole" che determinano un *reticolo* di insiemi.



3. L’architettura Web Services

Un “Web Service” può essere definito come un “sistema” la cui interfaccia pubblica è definita mediante una grammatica XML. Altri sistemi possono accedere alla definizione ed interagire scambiando messaggi XML, nelle modalità prescritte dalla definizione, trasmessi utilizzando un protocollo della Internet protocol suite.

L’architettura Web Services è orientata ai servizi. In questo ambito definisce tre ruoli distinti:

a) Service Requestor
richiede l’esecuzione di un servizio

b) Service Provider
processa le richieste di servizio

c) Discovery Agency
Rende pubblice le definizioni dei servizi.

Il “Service Provider” implementa un servizio e ne pubblica l’interfaccia presso una “Discovery Agency”. Il “Service Requestor” interroga la “Discovery Agency”, accedendo alla definizione del servizio, e si connette al “Service Provider” richiedendone l’esecuzione

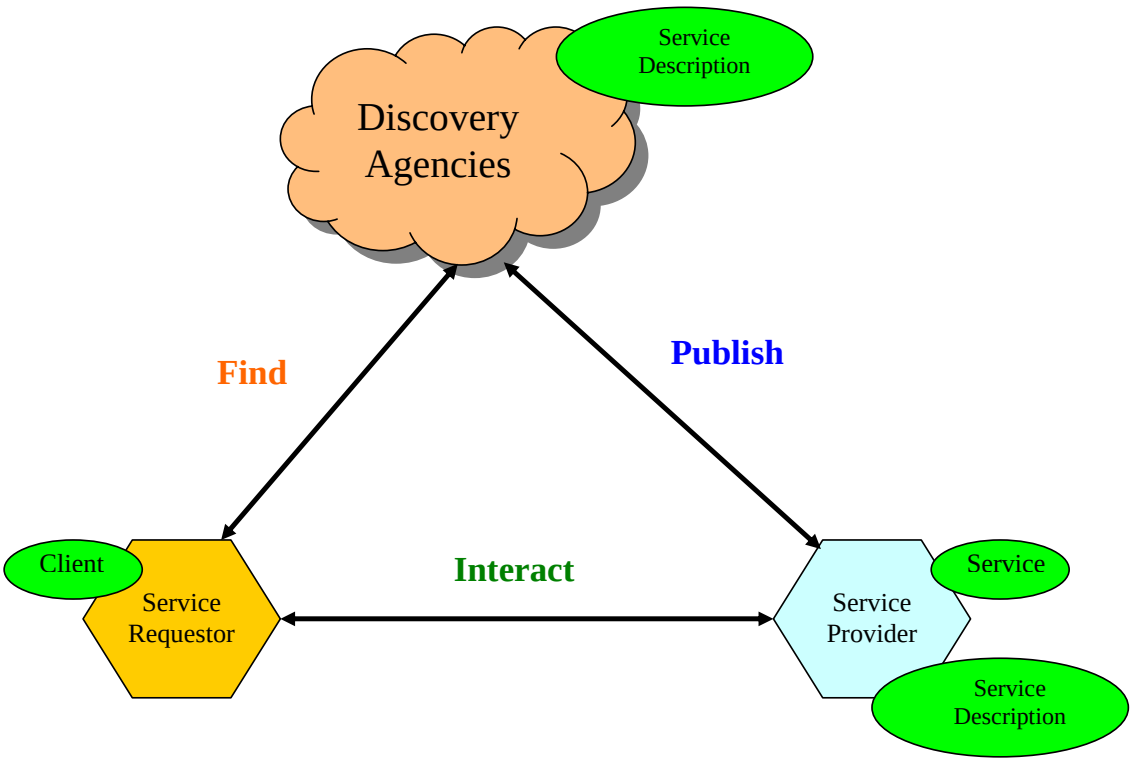
Una applicazione può assumere allo stesso tempo ruoli differenti, agendo sia come requestor, come provider o come discovery agency. Ogni singolo ruolo può interagire con gli altri secondo modalità ben definite:

a. Service Requestor
ricerca un servizio in una “Discovery Agency”
interagisce con un servizio di un “Service Provider”

b. Service Provider
pubblica il servizio presso una “Discovery Agency”
processa le richieste ricevute da un “Service Requestor”

c. Discovery Agency
pubblica i servizi registrati dai “Service Provider”
fornire la descrizione dei servizi ai “Service Requestor”.

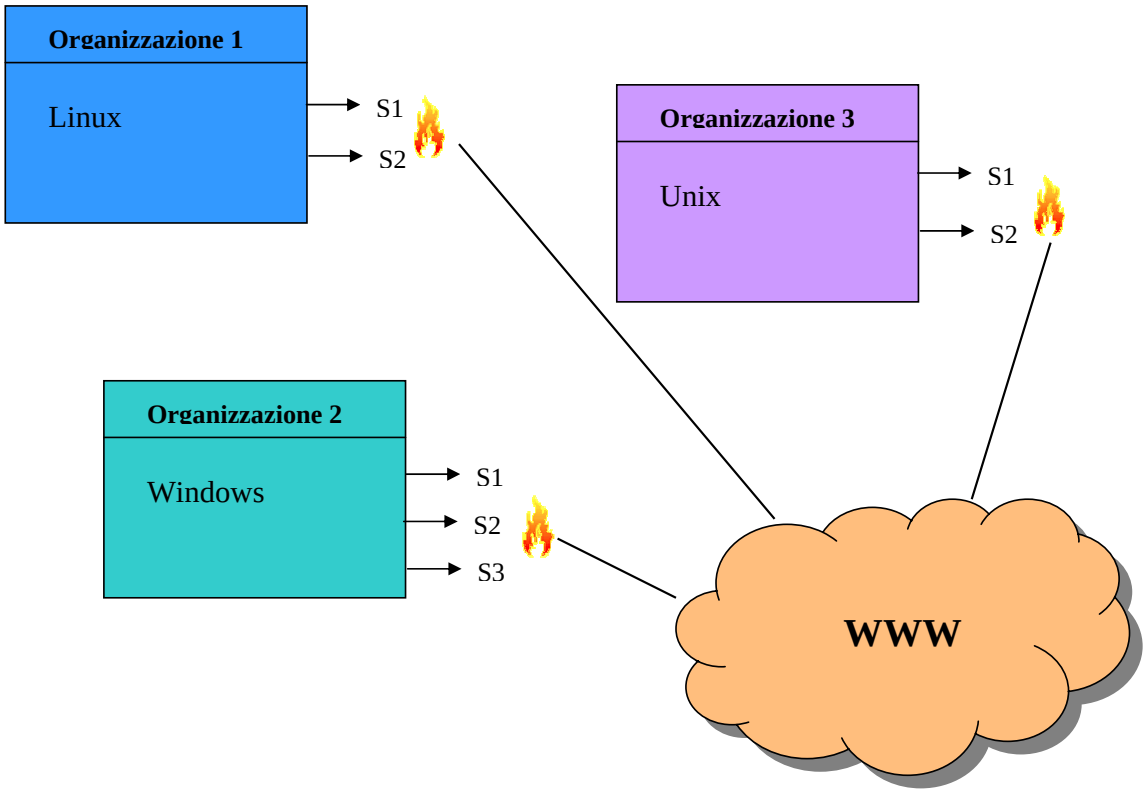
Un web service viene descritto usando una definizione “formale” mediante una grammatica XML, chiamata “Service Description”. La definizione comprende tutte le informazioni necessarie ad interagire con il sistema: formato dei messaggi, protocolli di trasporto e collocazione. L’interfaccia del serivizio “astrae” dai dettagli dell’implementazione ed è indipendente dal linguaggio di programmazione utilizzato nell’implementazione.



Un "**Web Service**" è quindi per definizione una interfaccia descritta mediante una "Service Description" la cui implementazione è "il servizio". Un servizio è un modulo software, accessibile attraverso funzionalità di networking, fornito da un Service Provider. Un servizio può a sua volta comportarsi come un Service Requestor utilizzando altri web services nella propria implementazione.

Una "**Service Description**" contiene la definizione dell'interfaccia del servizio ed nasconde le informazioni necessarie ad interagire con la implementazione del servizio. Include la definizione dei tipi di dati, le operazioni implementate dal servizio, le informazioni relative ad i protocolli necessarie per la connessione al servizio e gli indirizzi di rete che individuano la collocazione del servizio.

L'architettura Web Services pone le basi per la implementazione di sviluppo RPC adatto a risolvere le problematiche di interoperabilità descritte nella premessa. La situazione attuale dell'internetworking si sta ormai stabilizzando in uno scenario la cui struttura portante è definita dalla accessibilità al Word Wide Web. Le organizzazioni rendono disponibili l'accesso ai protocolli di base e, in generale, "pubblicano" porte di comunicazione standard: HTTP, SMTP, FTP, etc., mentre tendono a "sbarrare" qualunque altra forma di comunicazione, perché potenzialmente pericolosa.



I servizi esportati sono generalmente o di tipo “informativo”, documentazione, informazioni generali o servizi di base come posta elettronica, file transfer oppure servizi applicativi realizzati all’interno dell’organizzazione stessa e pubblicati mediante pagine web dinamiche.

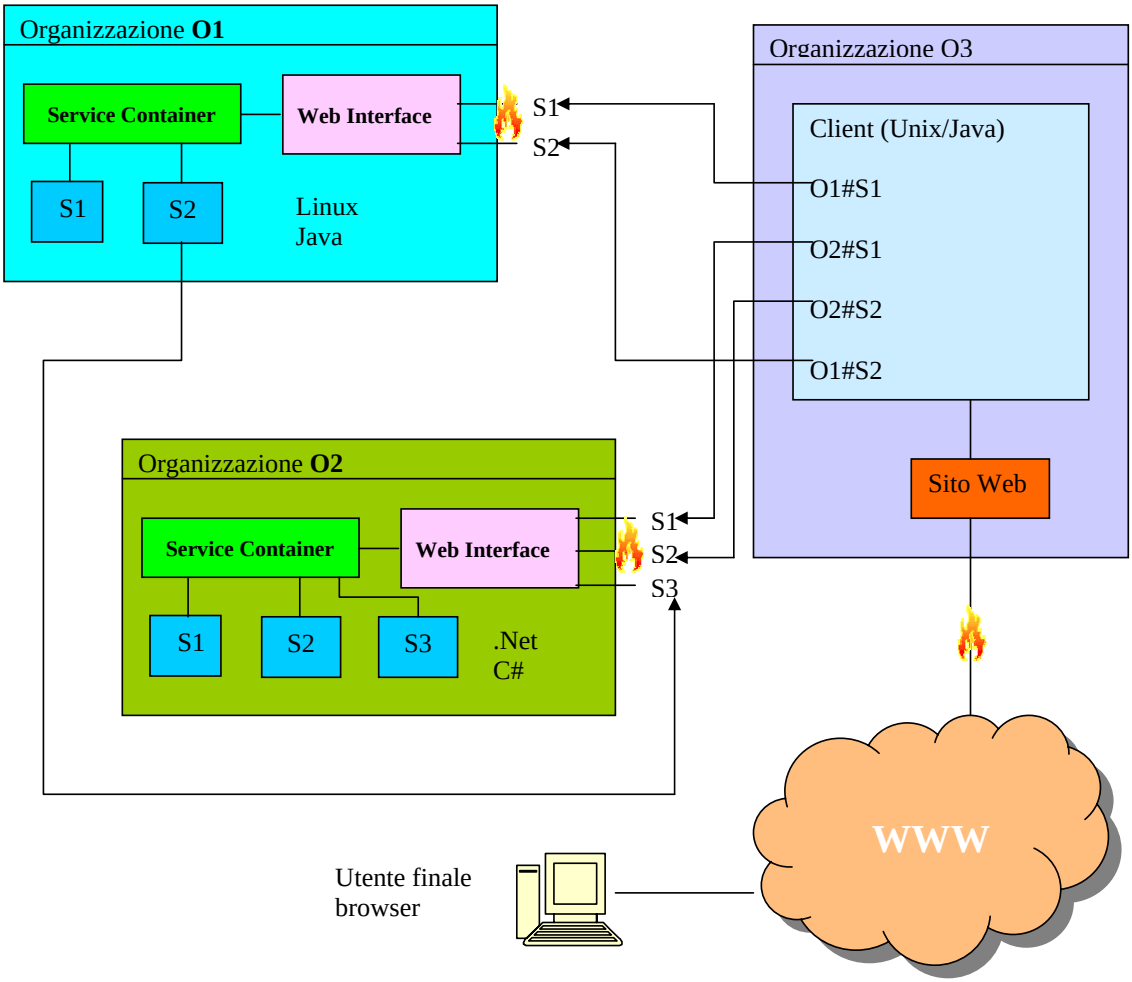
Il modello concettuale RPC viene generalmente utilizzato all’interno dell’organizzazione stessa, seguendo modalità che lo rendono intrinsecamente “incompatibile” con i servizi implementati da altre organizzazioni e spesso da altre unità organizzative dello stesso ente o della stessa azienda.

La tecnologia rende oggi possibile modellare la struttura organizzativa e le applicazioni che la descrivono seguendo nuove modalità. La necessità di “interoperare” per implementare servizi nella logica imposta dal modello B2B impone di ristrutturare le infrastrutture di networking e le applicazioni per poter perseguire nuovi obiettivi:

a) l’interfaccia e l’implementazione di un servizio devono essere largamente indipendenti;

b) l’interfaccia di un servizio deve essere “pubblica”, accessibile via rete e di fatto riutilizzabile;

c) deve essere possibile “comporre” i servizi in nuove applicazioni a loro volta offerte come servizi o direttamente fruite dagli utenti finali.



L'architettura Web Services rende possibile, nell'ambito del modello RPC, "scavalcare" la frontiera di una organizzazione, mantenendo le attuali infrastrutture di comunicazione e continuando ad utilizzare i protocolli di trasporto delle Internet protocol suite. I servizi possono quindi essere ricomposti per sviluppare nuove applicazioni offerte agli utenti finali in modo "trasparente".

Inoltre, nella tecnologia Web Services, un messaggio di richiesta non è automaticamente correlato con un messaggio di risposta, come nella maggior parte delle architetture RPC. La correlazione può avvenire a livello applicativo. Ciò rende, ad esempio, possibile utilizzare il protocollo SMPT come protocollo di trasporto: in questo caso i messaggi potrebbero essere scambiati via posta elettronica in modo asincrono.

La architettura Web Services trova il suo fondamento nelle tecnologie di networking: i sistemi web services sono acceduti, in generale, via rete, invocati da un Service Requestor. La sua universale disponibilità rende il protocollo HTTP particolarmente adatto a svolgere questa funzione, ma nulla vieta che altri protocolli, come SMTP, FTP, possano essere utilizzati come protocolli di trasporto. In applicazioni Intranet possono essere utilizzati anche protocolli "proprietari" come COM/DCOM, RMI, CORBA, etc.

4. Web Services Description Language: WSDL

Il Web Services Description Language (WSDL) è un linguaggio definito da una grammatica XML disegnata per astrarre la definizione delle funzionalità di un servizio web dai dettagli "concreti" della implementazione del servizio, come il "dove" e "in che modo" le funzionalità vengono offerte.

Il WSDL descrive *messaggi* che vengono scambiati tra service provider e service requestor, dove un *messaggio* è, per definizione un insieme di dati tipati (un intero, una stringa, etc.). Uno scambio di messaggi tra service provider e service requestor viene definito una *operazione*, un insieme di operazioni un *port type*. Un *servizio* è definito mediante un insieme di porte, dove ogni porta corrisponde all'implementazione di un port type. L'implementazione di un port type include tutti i dettagli concreti necessari ad interagire con il servizio.

Esiste uno stretto parallelo tra alcuni concetti della tecnologia object oriented e le definizioni introdotte nel WSDL. In una Service Description si possono identificare:

a) portType: un analogo della classe;

b) operazioni: modellate come i metodi di una classe;

c) messaggi: come gli argomenti dei metodi.

Le definizioni WSDL vengono rappresentate come documenti XML: il WSDL è per costruzione un sottolinguaggio dell'XML, definito mediante uno schema (http://www.w3.org/2002/ws/desc/): utilizza un formato XML per definire in modo preciso una Web Service Description in termini di:

1. **tipi** di dati
2. **messaggi**: insiemi di tipi di dati
3. **portType**: insiemi di messaggi, operazioni
4. **serviceType**: un insieme di porte correlate
5. **binding**: formato dei messaggi e protocolli utilizzati dal servizio
6. **service**: insieme di porte definite mediante un dato serviceType.

Un semplice esempio, che svilupperemo nel corso di questo documento, può essere utile a comprendere concretamente questi concetti. Possiamo provare a definire, in termini di linguaggio corrente, l'interfaccia pubblica di un *counter*, un contatore che mantiene un numero intero che può essere incrementato e decrementato. La definizione di questo semplice servizio può prendere le mosse da una semplicissima specifica:

Definizione del servizio **counter**:

- valore del contatore: un numero intero segnato
- metodi di accesso al servizio:
  - add (numero) aggiunge numero
  - subtract (numero) sottrae numero
  - getValue ( ) il valore del contatore

mediante l'interfaccia di un contatore.

Questa specifica può essere facilmente tradotta in qualunque linguaggio formale allo scopo di definire l'interfaccia della implementazione del contatore. Ad esempio in Java:

```
public interface Counter {

    public int add ( int val );

    public int subtract ( int val );

    public int getValue ( );
}
```

Non è difficile identificare in questa definizione l'interfaccia Counter come un portType, i metodi come operazioni, i valori ritornati e gli argomenti dei metodi come messaggi. Ed è altrettanto semplice *tradurre* questa definizione in WSDL. L'interfaccia Counter potrebbe essere definita con il portType:

```
<wsdl:portType name="CounterPortType">

   <wsdl:operation name="add" parameterOrder="in0">
      <wsdl:input message="intf:addRequest" name="addRequest"/>
      <wsdl:output message="intf:addResponse" name="addResponse"/>
   </wsdl:operation>

   <wsdl:operation name="subtract" parameterOrder="in0">
       <wsdl:input message="intf:subtractRequest" name="subtractRequest"/>
       <wsdl:output message="intf:subtractResponse" name="subtractResponse"/>
   </wsdl:operation>

   <wsdl:operation name="getValue">
       <wsdl:input message="intf:getValueRequest" name="getValueRequest"/>
       <wsdl:output message="intf:getValueResponse" name="getValueResponse"/>
   </wsdl:operation>
</wsdl:portType>
```

che definisce il CounterPortType come l'insieme di tre operazioni: add, subtract e getValue.
Ciascuna delle operazioni è a sua volta definita in termine dei messaggi scambiati tra requestor e provider. Ad esempio l'operazione di add è definita da:

```
<wsdl:operation name="add" parameterOrder="in0">
   <wsdl:input message="intf:addRequest" name="addRequest"/>
   <wsdl:output message="intf:addResponse" name="addResponse"/>
</wsdl:operation>
```

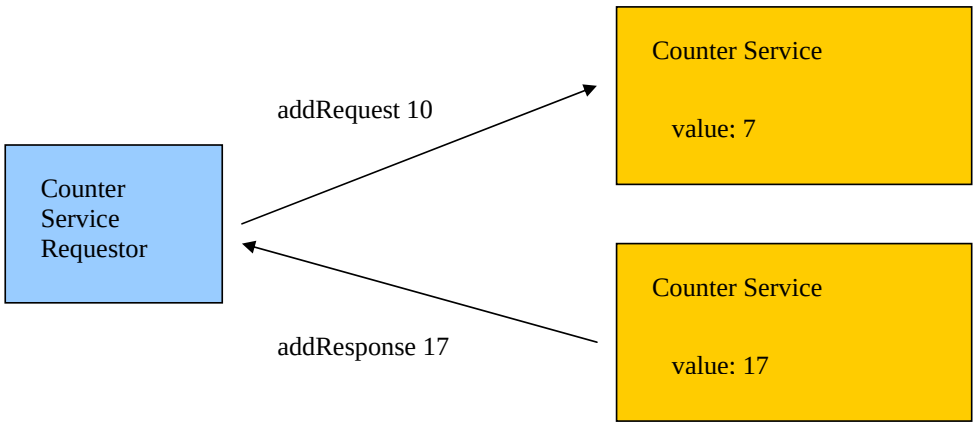
in termine di due messaggi: **addRequest** e **addResponse**. Entrambi i messaggi in questo caso, come definito dalla specifica consistono in un solo tipo di dati: un intero segnato. Il messaggio addRequest è un argomento in input al servizio e viaggia quindi dal requestor al provider, mentre il messaggio addResponse è un valore di ritorno in output e viene trasmesso dal provider al requestor.

La definizione dei messaggi è ancora una volta mutuata dall'XML. Ad esempio i messaggi addRequest e addResponse possono essere definiti come interi non segnati dal codice WSDL:

```
<wsdl:message name="addRequest">
    <wsdl:part name="in0" type="xsd:int"/>
</wsdl:message>

<wsdl:message name="addResponse">
    <wsdl:part name="addReturn" type="xsd:int"/>
</wsdl:message>
```

utilizzando il tipo di dati nativo XML ***xsd:int***.

Astraendo dal formato dei messaggi scambiati e dai protocolli utilizzati, il codice WSDL che abbiamo analizzato è tutto ciò che serve per definire una Service Description del servizio Counter che prescinde dal linguaggio utilizzato nella implementazione. In questo caso abbiamo preso le mosse da un frammento codificato come una interfaccia Java, ma è abbastanza semplice comprendere come la stessa cosa possa essere formalizzata utilizzando qualsiasi linguaggio di programmazione. Senza voler analizzare completamente il documento WSDL che definisce il portType del Counter è però semplice comprendere come l'interfaccia tra un requestor e un service provider che implementa un tale servizio definisce quanto necessario a descrivere l'interazione tra client e server, nei termini di un modello RPC.



Il Web Service Description Language viene utilizzato per costruire documenti WSDL che descrivono il servizio. In generale un Servive Requestor ricerca in una Discovery Agency, in una directory, il servizio a cui è interessato. Recupera quindi il file WSDL che lo descrive, una Web Service Description ed esegue il *parsing* del documento per recuperare le informazioni relative a:

- le coordinate, l'indirizzo, l'URI del servizio
- i metodi e gli argomenti del servizio
- come accedere i metodi: formato dei messaggi, protocolli di trasporto.

È a tutti gli effetti un "Interface Description Language" (IDL) del tutto analogo ai linguaggi utilizzati per descrivere le interfacce in architetture distribuite RPC come CORBA e DCOM.

La descrizione astratta non è ovviamente sufficiente al funzionamento di un servizio *concreto*: nella Service Description sono definite, sempre in linguaggio WSDL, le informazioni necessarie ad una *effettiva* comunicazione tra requestor e provider.

Il componente ***binding*** della descrizione si occupa dei dettagli relativi all'indirizzo, i protocolli, al formato del messaggio, etc.

Il componente ***service*** della descrizione assegna un nome al servizio ed elenca l'insieme dei portType che lo costituiscono:

```
<wsdl:service name="CounterPortTypeService">
   <wsdl:port binding="intf:calcSoapBinding" name="calc">
      <wsdlsoap:address location="http://localhost:8080/calc"/>
   </wsdl:port>
</wsdl:service>
```

in questo caso una porta SOAP (Simple Object Access Protocol).

La Service Description, nella forma di un file contenente un documento WSDL, può essere costruita manualmente, scritta da un essere umana o essere generata in modo automatico a partire da una descrizione *formale* in un linguaggio ad alto livello. Ad esempio, supponendo di aver compilato l'interfaccia Java dell'esempio nel file ***Counter.class*** è possibile ottenere il documento WSDL che descrive il servizio Counter usando il comando:

```
java org.apache.axis.wsdl.Java2WSDL -l http://localhost:8080/calc -P
        CounterPortType Counter
```

dell'AXIS Engine, che presuppone il formato SOAP dei messaggi ed utilizza il protocollo di trasporto HTTP.

5. Simple Object Access Protocol: SOAP

Il Simple Object Access Protocol, SOAP, è un protocollo definito da una grammatica XML e progettato per permettere lo scambio di messaggi in formato XML tra applicazioni. Il SOAP implementa un paradigma per lo scambio di *messaggi*, contenenti informazioni strutturate e tipate in un ambiente decentralizzato e distribuito. È un paradigma *stateless* (nel quale non vengono mantenute informazioni relative allo stato) per lo scambio di messaggi *unidirezionali* (one'way messages) mediante il quale le applicazioni possono creare interazioni *complesse*.

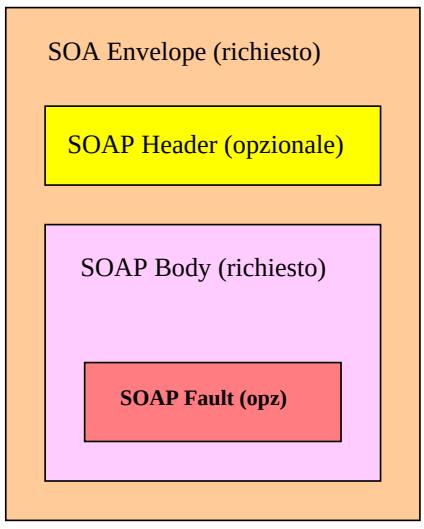
L'obiettivo principale del SOAP è centrato su un paradigma RPC, sulla chiamata a procedure remote il cui trasporto è garantito dal protocollo HTTP. Come abbiamo già fatto osservare, altre architetture come CORBA, DCOM e Java RMI implementano funzionalità analoghe, ma i messaggi SOAP sono definiti in termini di una grammatica XML e possono essere trasportati usando differenti protocolli di trasporto HTTP, SMTP, FTP, etc.

In questi termini, il particolare disegno del SOAP consente di superare con facilità i limiti imposti ad altre architetture dai sistemi orientati a garantire la sicurezza di rete che, in generale, tendono a bloccare il traffico dei protocolli specifici delle altre implementazioni RPC.

Un messaggio SOAP è fondamentalmente una comunicazione ***unidirezionale*** tra due *nodi SOAP*, un "**SOAP sender"** e un "**SOAP receiver"**, ma i messaggi SOAP possono essere ricombinati dalle applicazioni per implementare le interazioni complesse richieste da un paradigma RPC.

Un messaggio SOAP contiene una serie di elementi, alcuni dei quali obbligatori ed altri opzionali definiti dalle specifiche del protocollo:

- **SOAP Envelope**
  definisce il contenuto del messaggio
- **SOAP Header**
  estensioni, sicurezza, etc.
- **SOAP Body**
  contiene requests e responses delle chiamate RPC
- **SOAP Fault**
  messaggi relativi ad errori



Uno ***nodo*** SOAP implementa la logica necessaria processare un messaggio SOAP e può agire interpretando i ruoli di:

- **sender** il mittente del messaggio che dà origine alla comunicazione
- **relay** trasmette il messaggio ad un altro destinatario
- **receiver** riceve il messaggio e lo processa.

I nodi SOAP sono identificati univocamente mediante un **URI** (Uniform Resource Identifier), secondo le specifiche dell'RFC2396, nella prassi può essere identificato con un **URL**.

Un **messaggio** SOAP è l'unità di informazione che viene scambiata tra nodi SOAP, definita in termini di una opportuna grammatica XML. Il messaggio **deve** contenere un solo elemento *Envelope* che a suo volte deve contenere un elemento *Body* che in pratica rappresenta il contenuto semantico definito dall'applicazione.

Un esempio molto semplice di messaggio SOAP (eliminando le definizioni di schema e i namespaces), può essere reso schematicamente con:

```
<env:Envelope>
  <env:Body>
     <m:GetPrice>
       <Item>Lever2000</Item>
     </m:GetPrice>
  </env:Body>
</env:Envelope
```

Envelope

I messaggi SOAP devono contenere questo elemento come *radice* del documento XML. In pratica contiene le informazioni necessarie ad identificare la versione del protocollo, non i termini numerici, ma in termini di un namspace XML.

Header

L'elemento opzionale Header consente all'applicazione di estendere il protocollo aggiungendo informazioni addizionali relative ad esempio alla sicurezza, ad identificatori di transazione, etc.

Body

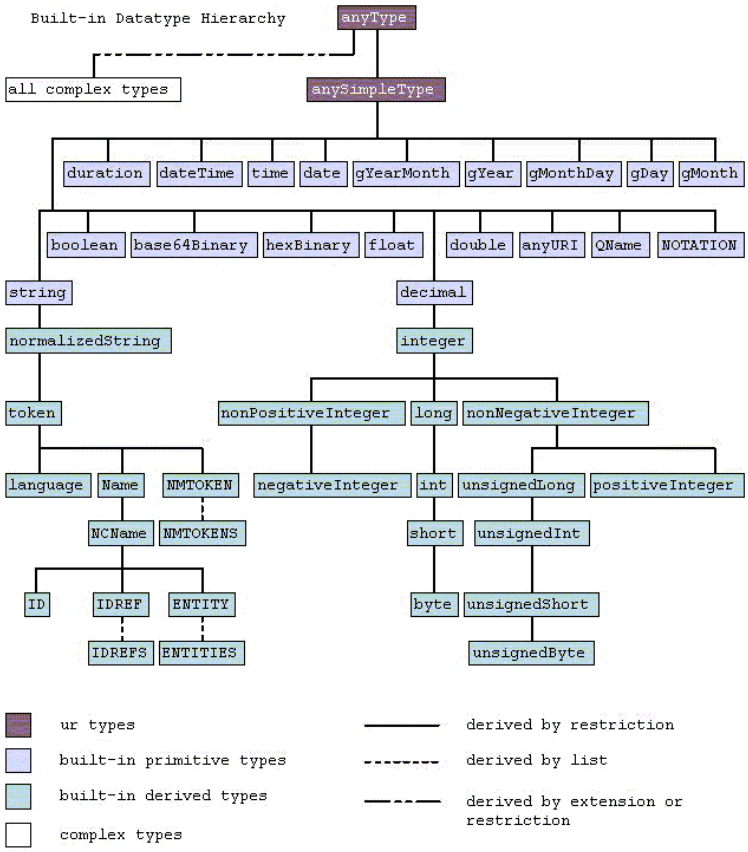
È un elemento obbligatorio che contiene **il messaggio** applicativo, nella maggior parte dei casi una richiesta o una risposta nei termini di un paradigma RPC.

Fault

Nel caso di errori l'elemento Body può contenere un elemento Fault in grado di descrivere il tipo di errore, la causa, di fornire insomma al requestor tutte le informazioni necessarie a gestire una situazione anomala.

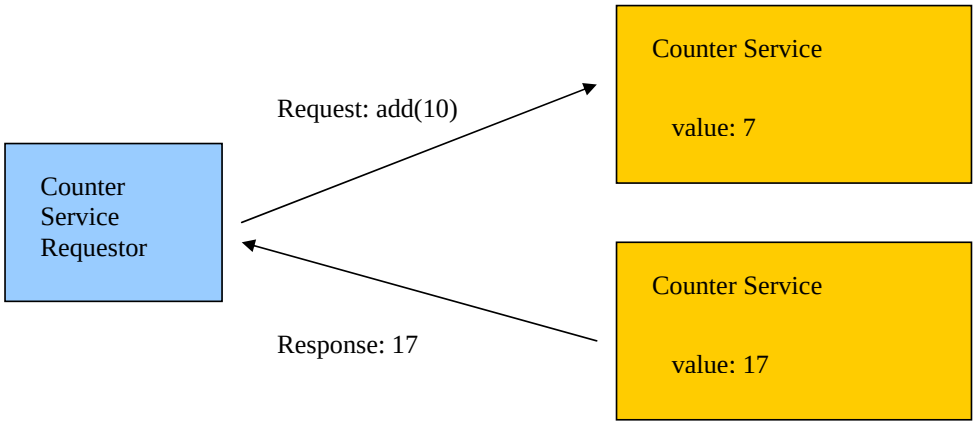
Come in tutti i paradigmi RPC il protocollo SOAP deve risolvere un problema fondamentale: architetture differenti codificano i dati in modi differenti. Anche solo trasmettere un numero intero da una macchina ad un'altra può spesso diventare un problema. Codificare le informazioni in modo che siamo *universalmente* interpretabili è un problema noto negli ambienti tecnici con il nome di *marshalling*/*unmarshalling*. I dati devono essere convertiti in un formato standard al momento della trasmissione e riconvertiti in formato *macchina* dopo essere stati ricevuti, affinché il destinatario del messaggio sia in grado di utilizzarli efficacemente.

Nel protocollo SOAP tale problema viene risolto facendo in parte riferimento alle definizioni della grammatica e degli schemi XML, infatti il soap mutua le definizioni dei tipi basi di dati per la definizione di uno schema XML. In particolare sono definiti i tipi dati più comuni:



che permettono la rappresentazione di interi, string, floating points, etc. Inoltre il protocollo SOAP prevede la rappresentazioni di tipi dati composti e di vettori unidimensionali.

Riprendendo l'esempio del nostro contatore, una interazione tra client e server, relativa al metodo ***add***, può essere rappresentata attraverso lo schema:



La richiesta ***add(10)*** viene inviata dal **nodo sender**, il client, al **nodo receiver**, il server utilizzando il messaggio SOAP:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:add soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="http://localhost:8080/axis/services/peppe/test1/Counter">
   <in0 xsi:type="xsd:int">10</in0>
  </ns1:add>
 </soapenv:Body>
</soapenv:Envelope>
```

che codifica la richiesta nel Body del messaggio SOAP. Da notare la conversione del tipo dati intero, argomento del messaggio add nel tipo dati XML corrispondente. In questo caso alla descrizione dell'interfaccia mediante il linguaggio WSDL è associato un messaggio XML che permette l'implementazione dell'interfaccia nel protocollo SOAP.

Nello stesso modo il *response* del server, previsto dall'interfaccia WSDL del counter, può essere trasmesso dal server al client codificato in un messaggio SOAP, ancora una volta codificato mediante un tipo dati intero XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:addResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="http://localhost:8080/axis/services/peppe/test1/Counter">
   <addReturn xsi:type="xsd:int">17</addReturn>
  </ns1:addResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Il protocollo SOAP non è legato ad un particolare protocollo di *trasporto*: SMTP, FTP, HTTP, etc. sono tutte scelte ugualmente praticabili. Per fare un esempio il protocollo SMTP (Simple Mail Transfer Protocol), normalmente utilizzato per trasferire messaggi di posta elettronica, potrebbe essere utilizzato dal SOAP come protocollo di trasporto, nei casi in cui un trasferimento di *batch* di richieste dovesse essere particolarmente adatto a risolvere un problema applicativo. Ciò nonostante il protocollo HTTP rimane il protocollo di trasporto più utilizzato.

Logicamente una richiesta SOAP viene inviata utilizzando una richiesta http e la risposta mediante un *response* HTTP. Il tipo di richiesta più utilizzato rimane la richiesta di tipo **PUT**, che non pone limitazioni sulla lunghezza del messaggio, ma nello stesso modo il messaggio SOAP può essere trasferito mediante un **HTTP GET**.

La richiesta ***add(10)*** inviata al counter potrebbe essere trasferita con un HTTP PUT come una richiesta HTTP del tipo:

```
POST /axis/services/Counter HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.1RC1
Host: 127.0.0.1
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Content-Length: 462
```

mentre la risposta del server sarebbe in questo caso un response HTTP del tipo:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Date: Mon, 14 Jul 2003 16:18:50 GMT
Server: Apache Tomcat/4.0.6 (HTTP/1.1 Connector)
Connection: close
```

In questo caso i messaggi SOAP undirezionali sono stati utilizzati per implementare il protocollo che implementa l'interfaccia WSDL del counter ed il protocollo HTTP è stato utilizzato per *trasportare* i messaggi tra client e server. Il risultato è una implementazione del paradigma RPC che permette al client di utilizzare una procedura remota, implementata come servizio mediante un opportuno server.

6. Implementazioni

I linguaggi ed i protocolli di cui abbiamo parlato stabiliscono un preciso riferimento per lo sviluppo di sistemi che seguono un paradigma RPC orientato ad applicazioni Web: fissano standard applicativi e protocolli di comunicazione che ne standardizzano struttura e modalità di sviluppo.

Ovviamente sono un riferimento "astratto": lo sviluppo di una applicazione "reale" può fondarsi solamente su una implementazione "concreta" di tali standard e su piattaforme applicative largamente diffuse.

Esistono diverse implementazioni WSDL/SOAP che consentono lo sviluppo di applicazioni Web Services basate su differenti piattaforme applicative. In particolare sono da citare l'ambiente J2EE, fondato sulla piattaforma Java, sviluppata da Sun e l'ambiente .NET della Microsoft.

Entrambi gli ambienti forniscono strumenti per sviluppare applicazioni Web Services su architetture Linux/Unix e Microsoft Windows, utilizzando differenti linguaggi di programmazione: Java, C, C++, C#, etc.

Per gli scopi di questa relazione analizzeremo in qualche dettaglio solamente una implementazione Open Source dell'ambiente J2EE, ma è comunque importante tenere presente che gli ambienti che fanno riferimento ai Web Services "dovrebbero" in ogni caso garantire la interoperabilità dei protocolli e più in generale delle applicazioni. Chiaramente una valutazione più precisa richiederebbe una analisi di dettaglio che va oltre gli scopi e le valutazioni che presenteremo in questa relazione.

L'implementazione che analizzeremo è Apache Axis, http://ws.apache.org/axis/: è un motore SOAP sviluppato utilizzando la tecnologia Java dal gruppo di sviluppo Apache. Axis è un "SOAP Engine" in grado di processare messaggi SOAP che stabilisce un framework applicativo per lo sviluppo di clients, servers e gateways SOAP. Axis Apache include:

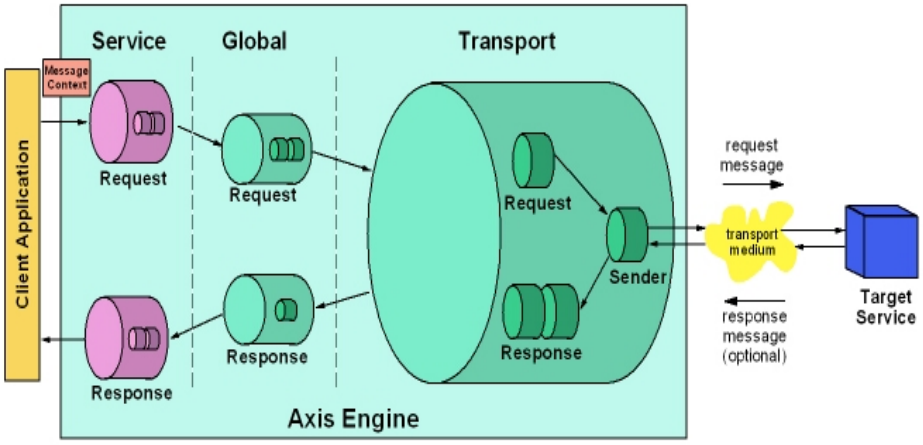
- un server stand-alone:
- un server utilizzabile in ambiente servlets (ad esempio Tomcat);
- supporto per il Web Services Description Language (WSDL);
- strumenti per la generazione automatica di classi java da codice WSDL;
- strumenti per il monitoring di messaggi SOAP.

È il successore di Apache SOAP, basato sull'IBM SOAP4J ed è stato rilasciato, nella versione 1.0 nell'Ottobre del 2002.

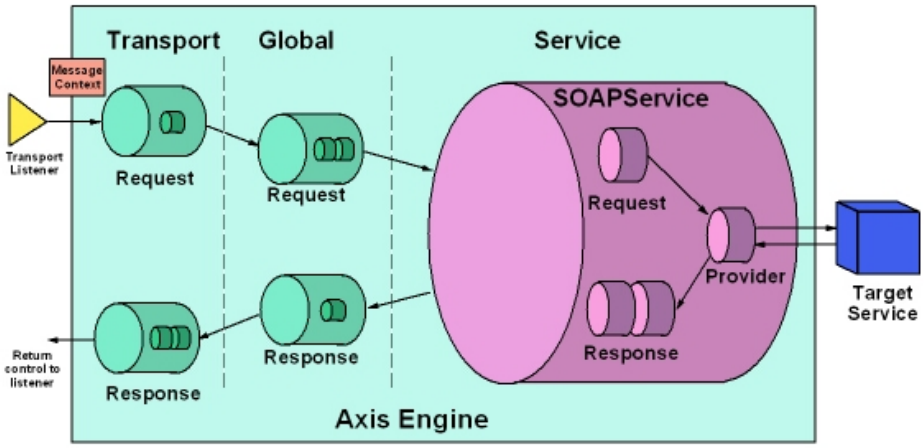
Semplificando Apache Axis fondamentalmente è in grado di processare messaggi, consegnandoli al servizio in grado di gestirli. I servizi vengono installati nel servente come classi Java ed il servente Axis è in grado di interpretare l'URL (nel caso di trasporto HTTP) della richiesta dirottandola nell'esecuzione della classe configurata per gestirla.

Axis può essere utilizzate in due modalità: server e client.

Dal lato "**client**" l'applicazione genera dei messaggi che vengono processati dalle librerie Axis per essere trasferiti al servizio in grado di gestirle (in generale un Web Service remoto).

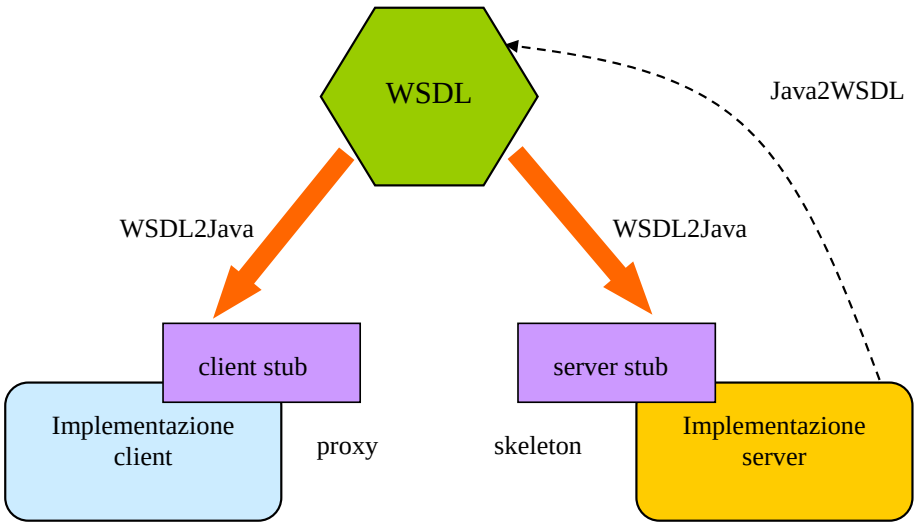


Dal lato "**server**" un messaggio raggiunge il livello di trasporto ed il servente Axis si occupa di identificare la classe Java che implementa il servizio.



Le implementazioni dei servizi vengono configurate al servente Axis utilizzando una apposita interfaccia di amministrazione ed installate come classi o librerie Java.

Axis fornisce gli strumenti necessari per la generazione automatica dei documenti WSDL dalla definizione Java delle interfacce e per la generazione del codice Java necessario alla implementazione del servizio ed al funzionamento del client.



Riprendendo l'interfaccia Java del contatore, che abbiamo definito parlando di WSDL,

```
public interface Counter {
    public int add ( int val );
    public int subtract ( int val );
    public int getValue ( );
}
```

possiamo utilizzare gli strumenti di sviluppo forniti da Apache Axis per sviluppare, a titolo di esempio, una implementazione del servizio ed un client in linguaggio Java.

Una volta compilato il sorgente dell'interfaccia, diciamo nel file ***Counter.class***, l'ambiente Axis mette a disposizione uno strumento, **Java2WSDL,** per convertire la definizione Java dell'interfaccia in un documento WSDL, ***Counter.wsdl***, del tutto analogo a quello che abbiamo già analizzato:

```
java org.apache.axis.wsdl.Java2WSDL -n example -l
http://localhost:8080/axis/services/Counter -o Counter.wsdl Counter
```

La routine Java2WSDL utilizza in input la classe compilata java e l'URI che verrà assunto dal servizio e produce in output la definizione WSDL del Web Service. Questo è tutto. È ovviamente sempre possibili codificare direttamente in WSDL il Service Description, ma un programmatore Java troverà sicuramente questa soluzione molto più comoda ed elegante.

Una volta definita l'interfaccia del Web Service e codificato il Service Description è possibile procedere alla implementazione del servizio. Il primo passo consiste nell'usare un altro strumento Axis, **WSDL2Java**, per generare le routine di interfaccia necessarie al funzionamento del lato server.

```
java org.apache.axis.wsdl.WSDL2Java -d Application -s -S true Counter.wsdl
```

Anche in questo caso l'uso di un solo comando, di una routine Axis, consente di generare automaticamente, nella directory example, tutto il codice sorgente Java necessario ad implementare il servizio. La routine interpreta la definizione WSDL del servizio e genera, sulla base dei parametri utilizzati, gli stub necessari alla implementazione di un servizio *persistente*, cioè di un oggetto le cui variabili di stato verranno preservate tra diverse invocazioni del servizio.

Tra i file generati, in questo caso, da WSDL2Java, il programmatore è fondamentalmente interessato ad un solo sorgente: ***CounterSoapBindingImpl.java***. In pratica si lmiterà a sostituire il codice "fittizio" generato da WSDL2Java con la implementazione delle funzioni previste dalla interfaccia Counter. Ed ecco nascere dalla definizione Java/WSDL l'implementazione del servzio Counter

```
package example;

import java.rmi.*;

public class CounterSoapBindingImpl implements example.Counter {
    private int val = 0;

    public int add ( int val ) throws RemoteException {
       this.val = this.val + val;
       return this.val;
    }

    public int subtract ( int val ) throws RemoteException {
       this.val = this.val - val;
       return this.val;
    }

    public int getValue ( ) throws RemoteException {
       return this.val;
    }
}
```

Il programmatore in pratica si è limitato ad intervenire scrivendo le linee di codice evidenziate in corsivo, ovvero ad implementare le funzioni richieste dalla definizione dell'interfaccia Counter.

WSDL2Java oltre a generare i server's stub genera anche una coppia di file ***deploy.wsdd*** e ***undeploy.wsdd***, ancora una volta dei documenti XML, noti in ambiente Axis come **Web Service Deployment Descriptor** (WSDD). Questi file possono essere utilizzati per configurare il servizio al servente Axis agendo con l'**AdminClient** sull'interfaccia di amministrazione del servente Axis.

Il primo esegue il "deploy" del servizio, ovvero lo rende disponibile e lo configura al motore Axis con il comando:

```
java org.apache.axis.client.AdminClient deploy.wsdd
```

mentre il secondo esegue l'undeploy del servizio, ovvero lo elimina dalla catena dei servizi disponibili:

```
java org.apache.axis.client.AdminClient undeploy.wsdd
```

Per finire di installare il servizio è sufficiente compilare i sorgenti Java e copiare le classi binarie Java nelle directory gestite dal servente Axis.

Non rimane altro a questo punto che codificare l'*utente* del servizio, il client. Anche in questo ci viene in aiuto la routine Axis WSDL2Java che, utilizzando un diverso insieme di parametri, partendo dal Service Description WSDL, genererà in modo automatico i client's stub: tutto il codice sorgente Java necessario alla implementazione del client, nel package *client*, appunto.

```
java org.apache.axis.wsdl.WSDL2Java -p client
http://localhost:8080/axis/services/Counter?wsdl
```

Da notare che in questo caso abbiamo fornito in input a WSDL2Java un URL: abbiamo supposto cioè che del servizio fosse già stato eseguito il deployment nel servente. Nel nostro esempio sarebbe stato ovviamente possibile utilizzare direttamente il documento WSDL, ma, in generale, il Service Description potrebbe risultare disponibile solo accedendo all'URI del servizio stesso. Ciò rende particolarmente agevole lo sviluppo in ambiente di networking: è possibile codificare il client del servizio partendo praticamente da scratch. L'URI del servizio, oltre a fornire l'accesso alla implementazione, permette di generare tutto il codice di interfaccia necessario alla implementazione del client.

Seguendo lo stesso schema del lato server, la routine WSDL2Java genererà nella directory client il package Java client ed al programmatore rimarrà solamente da importare il package nella propria applicazione.

Nel nostro esempio supporremo che tutta l'applicazione si riduca a sommare 10 e sottrarre 3, per procedere quindi alla visualizzazione del valore del contatore (che e' inizialmente posto a zero dalla implementazione del server).

```
import client.*;

public class Client {
    public static void main ( String[] args )  throws Exception {
            CounterServiceLocator l = new CounterServiceLocator();
            Counter counter = l.getCounter();

            counter.add ( 10 );
            counter.subtract ( 3 );

            int value = counter.getValue();

            System.out.println ( value );
    }
}
```

La classe **Client**, nel nostro esempio, è in pratica l'utente del servizio Counter. Una volta compilata, alla prima esecuzione, l'applicazione si limiterà a ritornare il valore 7. Le successive esecuzioni, data la persistenza del servizio, ritorneranno valori multipli di 7.

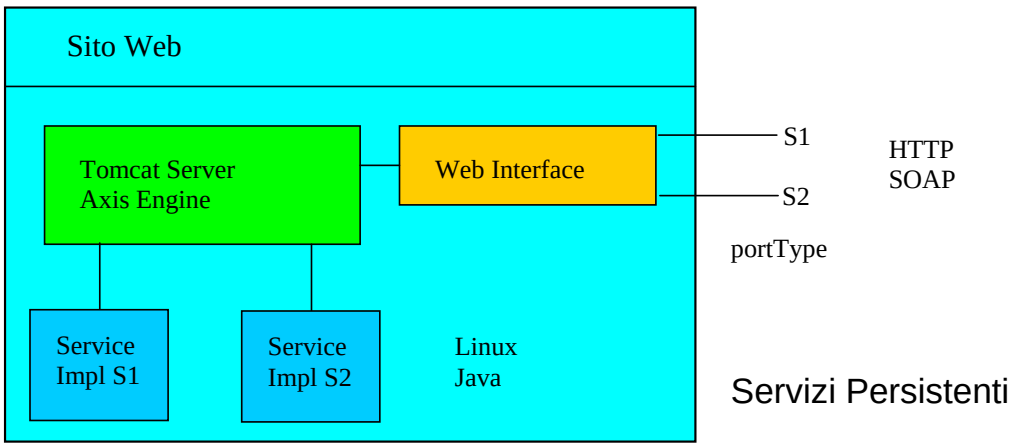
Il codice della classe client mostra quanto sia semplice programmare l'applicazione. Le routine di stub, generate per il client da WSDL2Java, si occupano di fornire all'applicazione una istanza della classe Counter, l'oggetto "**counter**" che è sempre possibile utilizzare come se fosse un oggetto locale dell'applicazione, invocando i metodi definiti nella interfaccia della classe Counter.

Il paradigma RPC è pienamente realizzato: dalla Service Description Java/WSDL del Web Service siamo stati in grado di codificare sia una implementazione del servizio, sia una applicazione che richiama, in modo praticamente trasparente, i metodi dichiarati nell'interfaccia.

Axis Engine implementa il linguaggio WSDL 1.1 (1.2 in elaborazione) e il protocollo SOAP 1.1/1.2: il paradigma RPC viene realizzato riferendosi alla definizione astratta che abbiamo analizzato nelle prime sezioni di questa relazione.

Dal paradigma "astratto" siamo passati ad una implementazione "concreta":

- Open Source
- ambiente di programmazione Java J2SE
- compatibile con lo standard J2EE
- una delle possibili.



7. Griglie Computazionali

Il calcolo ad alte prestazioni pone da sempre le sfide più avanzate alla tecnologia dell'informazione. In questo ambito sono nate le tecnologie di internetworking e il World Wide Web: il mondo del calcolo scientifico sente come esigenza primaria la necessità di scambiare ed elaborare informazione su piattaforme parallele e distribuite.

Da questa esigenza si è venuto evolvendo il paradigma della **Griglia Computazionale**: uno strumento per distribuire il calcolo ad alte prestazioni, per la gestione e la condivisione di risorse geograficamente distribuite e con l'obiettivo primario di rendere *trasparente* la complessità all'utente finale, in modo sicuro ed affidabile.

La complessità delle piattaforme hardware e software è infatti andata crescendo nel corso dell'evoluzione delle tecnologie informatiche e telematiche; se fino a qualche tempo gli sviluppatori potevano fare affidamento su piattaforme ed ambienti omogenei, sicuri, affidabili e gestiti da un'unica organizzazione, ora lo scenario presenta vincoli e possibilità completamente differenti.

L'attenzione si sta spostando sulla interconnessione di sistemi sia all'interno che all'esterno di una unità organizzativa. Inoltre aziende ed enti spesso trovano vantaggi significativi nell'affidare a fornitori esterni, attraverso meccanismi di *outsourcing*, parti non essenziali delle proprie attività informatiche.

Le applicazioni moderne vengono sviluppate ed ospitate su piattaforme eterogenee (Windows, Unix, Linux, J2EE, Microsoft .NET) sulle quali sono disponibili una grande varietà di servizi di base espressi in termini di funzioni di gestione delle risorse, database, clustering, sicurezza, gestione del carico e diagnostica hardware e software. Tali servizi sono spesso implementati con modalità molto differenti sia in termini di API che di semantica di funzionamento.

Per affrontare queste nuove esigenze è necessario sviluppare un livello di astrazione che consenta di assemblare dinamicamente risorse messe a disposizione da differenti unità organizzative, siano esse enti, aziende o fornitori di servizi.

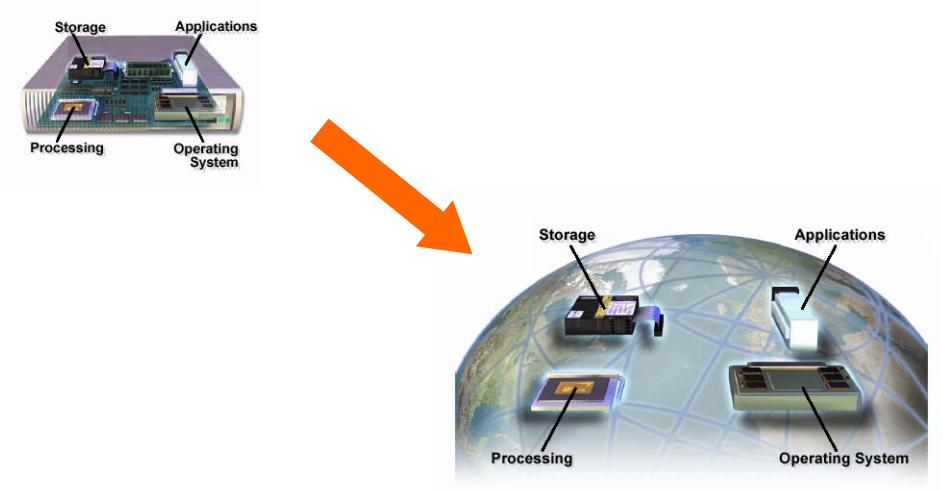
Questo problematiche sono state, negli ultimi anni, al centro dell'attenzione degli sviluppatori di applicazioni distribuite: il paradigma iniziale della "macchina rete", della rete come sistema intelligente, si è dovuto confrontare con un sempre più elevato livello di dispersione dei servizi. Dove prima un numero limitato di server, gestiti centralmente e interconnessi in unica intranet, erano sufficienti a gestire le applicazioni, ora ci si trova spesso a dover gestire la complessità generata da un gran numero di servizi, implementati su piattaforme eterogenee, che interagiscono attraverso reti "insicure" gestite da differenti fornitori.

L'ambiente scientifico del calcolo ad altre prestazioni, d'altra parte, si trova ad affrontare la stessa sfida: la necessità di strette collaborazioni tra differenti enti di ricerca, di interazione tra sistemi di calcolo e di gestione, di controllare remotamente complesse apparecchiature analogiche e digitali pone lo stesso tipo di problematiche. Assistiamo ad una convergenza tra le esigenze tecnologiche del mondo scientifico e degli ambienti degli affari e della Pubblica Amministrazione.

D’altra parte stiamo anche assistendo ad una evoluzione dell’hardware e del software di base nella direzione della distribuzione delle funzionalità di base di un calcolatore.
Parafrasando una citazione di George Gilder

> “*When the network is as fast as the computer's internal links, the machine disintegrates across the net into a set of special purpose appliances.*”

man mano che le reti di telecomunicazione divengono veloci come i canali di comunicazione interni di un computer, le macchine tendono a distribuirsi su Internet in una serie di periferiche specializzate.



Si tratta di uno scenario limite, che forse non potrà mai essere completamente realizzato a causa delle limitazioni imposte alla latenza dalla velocità della luce, ma che rende molto bene l’idea di come le funzioni primarie, che oggi siamo abituati a pensare come appartenenti ad una stessa macchina, possano distribuirsi su una rete geografica di dimensioni mondiali.

Il paradigma di “Griglia Computazionale” prende lo spunto dalle griglie per la distribuzione dell’energia. In questa visione, macchine con compiti specifici potranno essere connesse alla griglia per la distribuzione e la trasformazione delle informazioni con la stessa facilità con cui oggi connettiamo un dispositivo alla rete per la distribuzione dell’energia.

Partendo da questa *visione* le tecnologie di Griglia si pongono come obiettivo la creazione di “Organizzazioni Virtuali”, multi istituzionali, per l’accesso a risorse di calcolo, dati e servizi mediante la creazione di funzionalità ottenute dalla composizione di servizi distribuiti.

Gli scenari nei quali questo paradigma può essere applicato possono essere molto differenti.

Nel campo delle applicazioni scientifiche, e-Science, alcuni esempi possono essere:

- biochmici che analizzano 100.000 composti in poche ore, utilizzando i cicli idle di decine di migliaia di computers;
- esperimenti di fisica che richiedono petaflops per analizzare petabytes di dati distribuiti a livello mondiale;
- climatologi che visualizzano ed analizzano terabytes di dati;
- chimici teorici che modellano e simulano grandi sistemi nel campo delle scienze molecolari e delle nano tecnologie.

Le aziende commerciali d'altra parte, e-Business, possono altresì applicare lo stesso modello:

- ingegneri di una multinazionale che collaborano nella progettazione di un nuovo prodotto;
- analisi multidisciplinari nel campo aerospaziale che condividono applicazioni e dati;
- compagnie di assicurazione che compiono analisi con tecnologie di data mining;
- fornitori di servizi, nel settore delle telecomunicazioni, che mettono a disposizione potenza di calcolo altresì inutilizzata.

L'amministrazione pubblica, e-government, si configura a sua volta come un insieme di enti, con un'ampia autonomia amministrativa, che si presentano sul Web come entità autonome che utilizzano tecnologie eterogenee e potenzialmente incompatibili.

Le tecnologie di griglia potrebbero essere applicate per:

- migliorare l'efficienza operativa interna della Pubblica Amministrazione;
- informatizzare l'erogazione dei servizi ai cittadini ed alle imprese;
- consentire l'accesso telematico ai servivi della Pubblica Amministrazione e la loro interoperabilità;
- condividere le informazioni;
- automatizzare i processi di cooperazione.

8. Globus Toolkit

Il paradigma “Griglia Computazionale” si è andato evolvendo negli ultimi tre anni, tra il 2000 ed il 2003, attraverso l’implementazione di varie tecnologie a progetti su scale a volte molto differenti. Una delle tecnologie di più ampia diffusione proviene dal progetto Globus [http://www.globus.org/] ed ha dato origine a **Globus Toolokit**.

La piattaforma Globus, sponsorizzata dal governo americano (DARPA, DOD, NSF) e da alcune tra le più importanti multinazionali americane (IBM, Microsoft e Cisco), è una architettura aperta, basata su tecnologia *open source* ed implementa una serie di librerie di supporto per applicazioni di griglia.

Il **toolkit** nasce con l’obiettivo di fornire una risposta alle esigenze di:

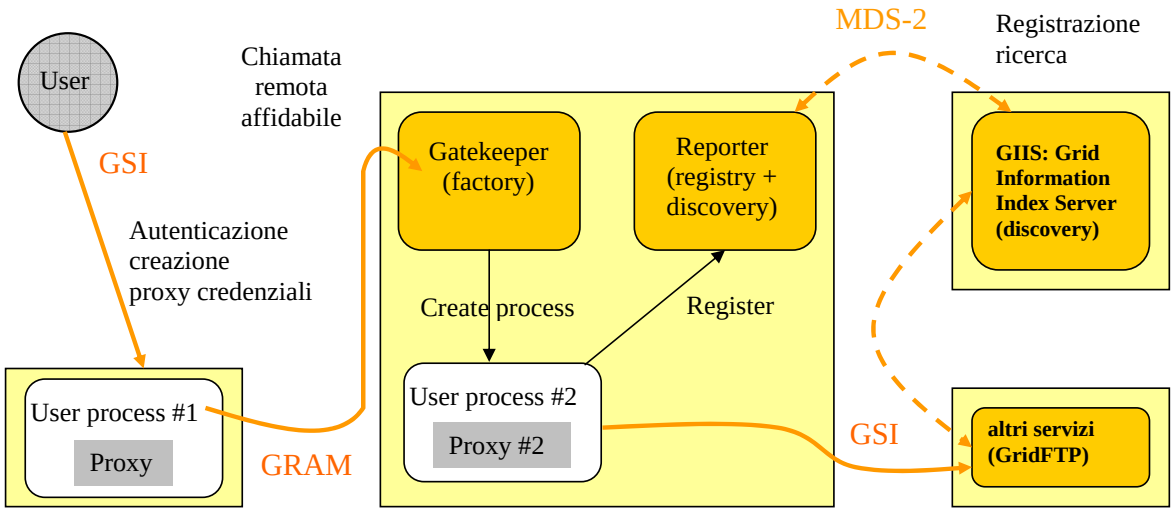
- sicurezza
- individuazione delle risorse
- gestione delle risorse
- gestione dei dati
- comunicazione
- gestione dei malfunzionamenti
- portabilità

poste dall’ambizioso obiettivo di implementare una griglia computazione di uso generale.

I componenti di interesse nel Globus Toolkit sono:

- il protocollo **GRAM** (Grid Resource Allocation and Management) ed il servizio **Gatekeeper** che forniscono un sistema di gestione sicuro ed affidabile per la gestione dei servizi;
- l’**MDS** (Meta Directory Service) per la registrazione e l’individuazione dei servizi;
- **GRAM Reporter**: una implementazione di registro “locale” che coopera con MDS;
- **GSI** (Grid Security Infrastructure), un sistema di credenziali di griglia per l’autenticazione attraverso meccanismi di single sign on e di delega;
- **Data Grid**: tecnologie per il trasferimento (GridFTP), la replica e la gestione dei dati;
- **API** (Application Programming Interface): l’interfaccia applicativa di Globus.

Questi componenti forniscono gli elementi essenziali di una architettura di griglia orientata ai servizi:



Attraverso il protocollo GRAM è possibile creare e gestire servizi *transitori* per la gestione di task orientati alla computazione mentre la tecnologia GSI viene utilizzata per l'autenticazione degli utenti e la gestione delle autorizzazioni, attraverso meccanismi di delega delle credenziali.

La creazione di nuovi servizi è gestita da un processo, il Gatekeeper, ed il GRAM reporter è responsabile di pubblicate le informazioni relative allo stato dei servizi *locali*, implementando una sorta di registro delle computazioni in corso nel nodo GRAM.

Tali informazioni vengono pubblicate attraverso l'infrastruttura fornita dall'MDS-2, un sottosistema che consente l'accesso alla configurazione dei server computazionali, allo stato della rete ed alla posizione dei datasets replicati.

Il protocollo GSI, basato su un sistema a chiave pubblica, implementa un meccanismo di autenticazione che consente il *single sign on*, ovvero, permette all'utente finale di autenticarsi una sola volta e di delegare quindi tale autenticazione ai servizi che istanziati su sua richiesta. L'infrastruttura GSI utilizza certificati X.509, secondo lo standard dei certificati PKI, come base per l'autenticazione.

La versione 2.4 di Globus Toolkit implementa i protocolli descritti in questa breve presentazione ed offre un ambiente di sviluppo che offre gli elementi essenziali della Griglia Computazionale, un ambiente ormai largamente utilizzato nell'ambito del calcolo ad altre prestazioni.

9. OGSA: Open Grid Service Architecture

Globus Toolkit versione 2 rappresenta una buona base tecnologica nel percorso dell'implmentazione di Griglie Computazionali. Fornisce le infrastrutture necessarie all'autenticazione degli utenti, all'individuazione ed al monitor delle risorse, all'invocazione affidabile di servizi remoti ed all'accesso ai dati.

Dall'altra parte i Web Services implementano una paradigma di calcolo distribuito che presenta degli indubbi vantaggi in termini di supporto industriale e di adesione agli standard utilizzati su Internet.

Entrambe le piattaforme presentano delle opportunità che sono state prese in considerazione nella definizione, da parte del progetto Globus, della **Open Grid Service Architecture**, una architettura aperta che ha come scopo la definizione della griglia computazionale come griglia di servizi.

La **O**pen **G**rid **S**ervice **A**rchitecture (**OGSA**) definisce i servizi in termini di Web Services ed utilizza i componenti implementati nell'ambito del Globus Toolkit. Il dettaglio di tale definizione, la base tecnologica della OGSA, è nota come **O**pen **G**rid **S**ervice **I**nfrastructure (**OGSI**).

Nella OGSA, per definizione, un **Grid Service** è un Web Service in cui è definito almeno il portType ***GridService***, una interfaccia che offre le primitive necessarie per individuare, creare dinamicamente e gestire un servizio.

Tutti i Grid Service *devono* implementare questa interfaccia, una condizione in generale non soddisfatta da un Web Service del tutto generale nel quale le interfacce offerte dipendono dall'applicazione.

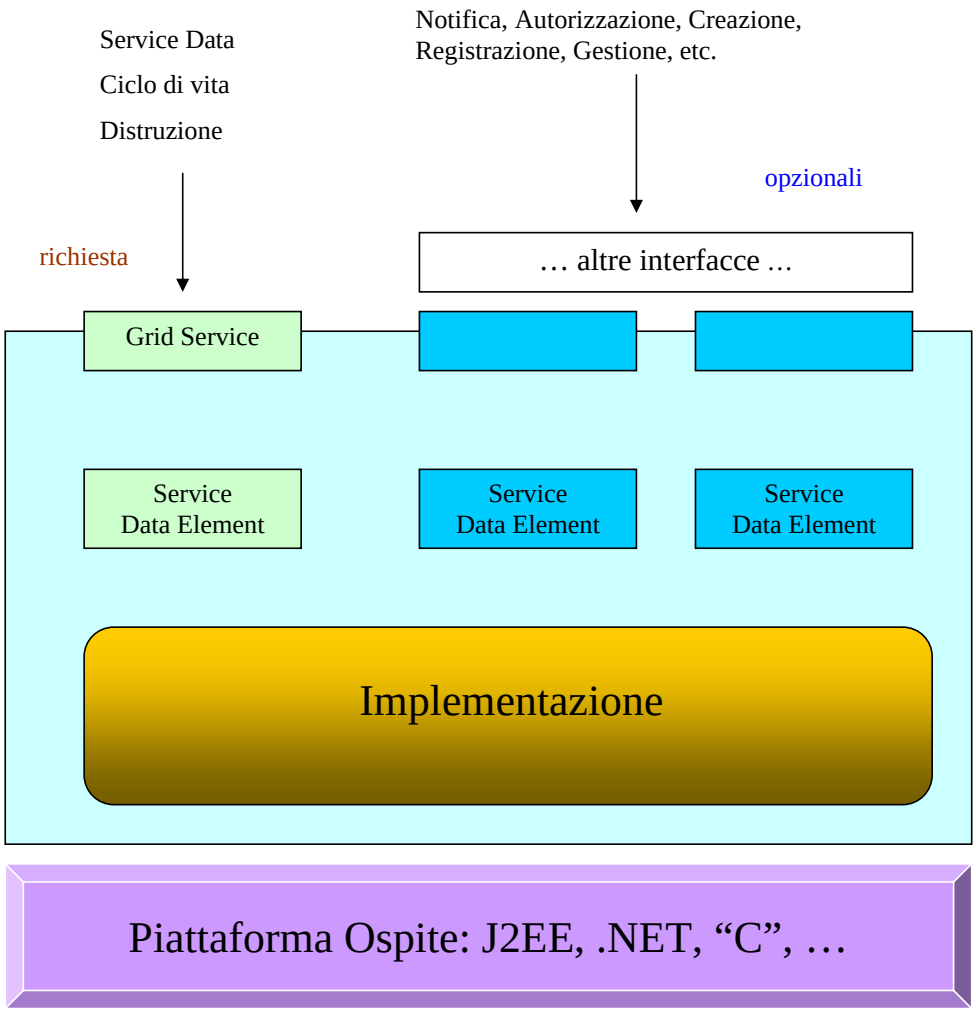
Nell'architettura OGSA viene fatta una distinzione tra una Grid Service *description*, una prescrizione sulle modalità con cui si interagisce con il servizio ed una *istanza* del Grid Service, un particolare servizio che implementa le interfacce definite dalla descrizione.

Le istanze dei Grid Service possono essere persistenti o transitorie ed essere, opzionalmente, accompagnate da elementi **serviceData** (**SDE** Service Data Element), informazioni aggiuntive relative all'istanza (*meta-data*) o a proprietà dell'istanza stessa che possono variare a runtime (*state-data*).

Il portType GridService implementa un ben definito insieme di operazioni:

- **FindServiceData**: accede gli elementi serviceData di una istanza;
- **RequestTermination**: accede e modifica l'istante in cui una istanza verrà terminata;
- **Destroy**: richiede la distruzione di una istanza del servizio.

Questo insieme di operazioni, con i relativi messaggi, deve essere implementato affinché si possa parlare di Grid Service nei termini della archiettura OGSA.



Nella definizione OGSI sono state introdotte delle estensioni WSDL per permettere l’implementazione dei Grid Services:

- **serviceData**: proprietà del servzio *osservabili* dall’esterno;
- **serviceDataDescription**: descrizione formale degli elementi serviceData;
- convenzioni sui nomi dei portType;
- **GSR**: Grid Service Reference;
- **GSH**: Grid Service Handle.

L'implementazione della architettura OGSA che prenderemo a riferimento è Globus Toolkit 3.0, disponibile all'URL http://www.globus.org. La piattaforma ospite è una macchina virtuale Java 2 Standard Edition J2SE. L'ambiente di riferimento contiene l'infrastruttura di Globus Toolkit 2.0 e una serie di strumenti che citiamo a scopo di riferimento:

- J2SE 1.3/1.4
- Jakarta Ant 1.5
- Jakarta Tomcat 4.1
- Apache Axis 1.1
- Apache Xerces 2.4

e le librerie che contengono l'implementazione dei Grid Service secondo lo standard definito dalla OGSI.

È disponibile anche un ambiente Microsoft .NET che fa riferimento a Microsoft .NET Framework 1.1 e a Visual Studio .NET 2003 che non analizzeremo in questa relazione.

A scopo di esemplificazione riprendiamo la definizione del *contatore* che abbiamo analizzato nell'implementazione Web Service. Il sorgente Java che abbiamo analizzato può facilmente essere riutilizzato per definire l'interfaccia del servizio in termini di Grid Services:

```
package gvt.sample.impl;

public interface GCounter {
    public int add ( int val );
    public int subtract ( int val );
    public int getValue ( );
}
```

che rappresenta un buon punto di partenza nell'implementazione del GridService **GCounter**.

Utilizzando gli strumenti OGSA del Globus Toolkit 3.0 è relativamente semplice generare automaticamente la descrizione WSDL del servizio in una forma adatta ad essere utilizzata nell'ambiente GT3. I metodi Java:

```
org.apache.axis.wsdl.Java2WSDL

org.globus.ogsa.tools.wsdl.DecorateWSDL
```

svolgono esattamente questo compito. Come nel caso dei Web Services, le librerie necessarie all'implementazione del servizio e delle applicazioni, possono essere generate automaticamente dalla routine: **org.globus.ogsa.tools.wsdl.GSDL2Java**.

Allo sviluppatore non rimane altro da fare che implementare il servizio, scrivendo il codice Java necessario alla funzionalità, in questo caso molto semplice, del servizio di griglia.

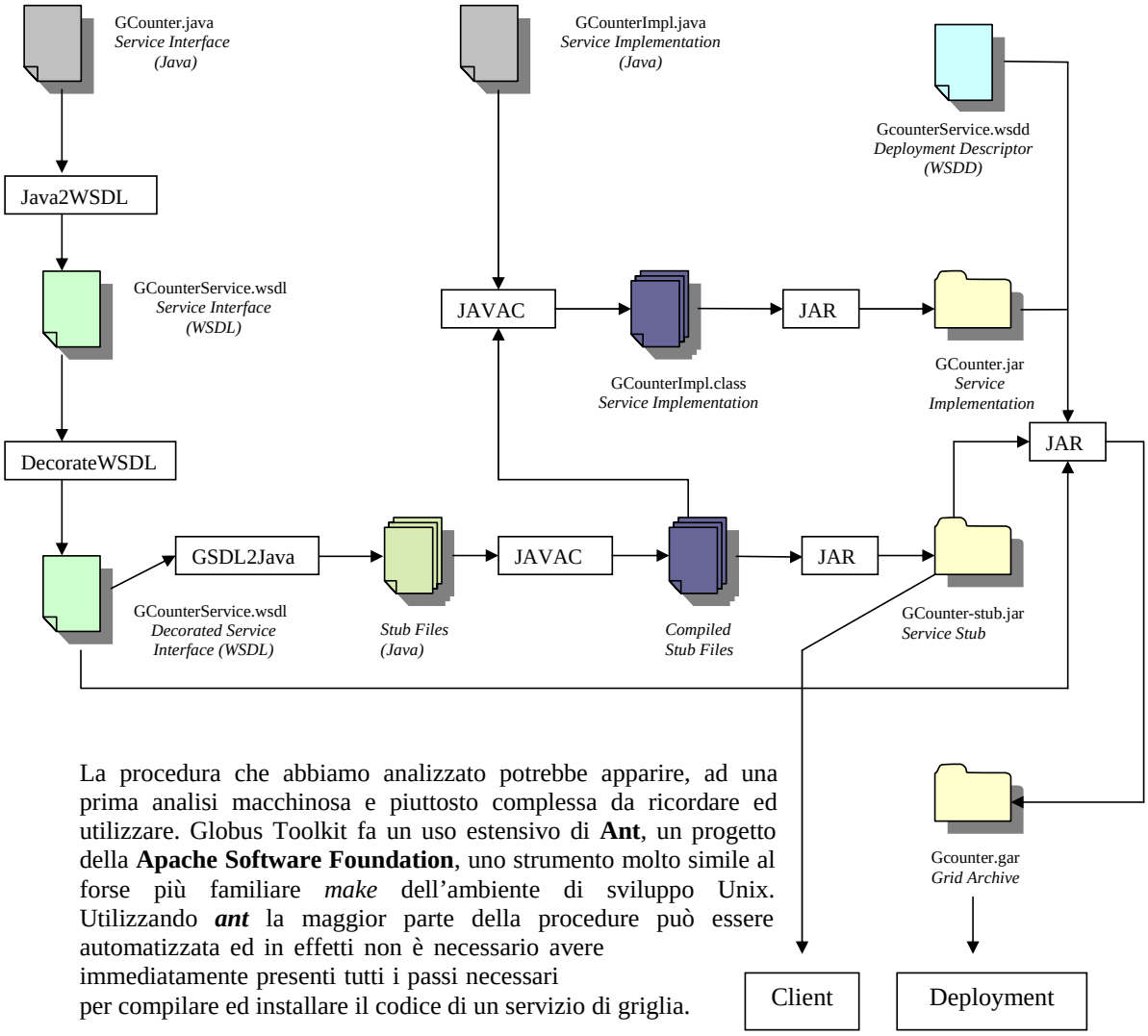
```
package gvt.sample.impl;

import org.globus.ogsa.impl.ogsi.GridServiceImpl;
import gvt.sample.GCounter.GCounterPortType;
import java.rmi.RemoteException;

public class GCounterImpl extends GridServiceImpl
    implements GCounterPortType {

    private int val = 0;

    public GCounterImpl() {
        super ( "GCounter" );
    }

    public int add ( int val ) throws RemoteException {
        this.val = this.val + val;
        return this.val;
    }

    public int subtract ( int val ) throws RemoteException {
        this.val = this.val - val;
        return this.val;
    }

    public int getValue ( ) throws RemoteException {
        return this.val;
    }
}
```

Anche in questo caso il programmatore si è limitato a scrivere il codice necessario a fornire le funzionalità *prescritte* dalla interfaccia del servizio, realizzando in effetti il server necessario ad una applicazione che dovesse utilizzare il Gcounter.

Un *deployment descriptor*, concettualmente analogo a quello dell'ambiente Axis, permetterà di installare il Grid Service, utilizzando in effetti il comando **ant** per semplificare le operazioni ripetitive.

Compilare ed installare un servizio di Griglia può, ad una prima analisi apparire complesso e macchinoso: uno schema può forse risultare di aiuto nella comprensione della procedura e degli strumenti Globus Toolkit utilizzati durante l'implementazione del contatore come servizio di griglia.

Dall'interfaccia, codificata in Java, si passa ad una classe compilata e quindi al portType WSDL. Il codice WSDL viene quindi utilizzato per generare gli stub files di interfaccia (codice sorgente Java). Gli stub, opportunamente compilati, costituiranno le classi Java di interfaccia che permetteranno di codificare l'implementazione del Grid Service. Una volta archiviate, mediante il comando **jar** (molto simile nell'uso al comando tar Unix), le classi stub e l'implementazione del servizio, il deployment descriptor (in formato WSDD), fornirà le informazioni necessarie ad archiviare il codice eseguibile in un Grid Archive (.gar), che potrà essere distribuito ed installato con estrema semplicità.



La procedura che abbiamo analizzato potrebbe apparire, ad una prima analisi macchinosa e piuttosto complessa da ricordare ed utilizzare. Globus Toolkit fa un uso estensivo di **Ant**, un progetto della **Apache Software Foundation**, uno strumento molto simile al forse più familiare *make* dell'ambiente di sviluppo Unix. Utilizzando ***ant*** la maggior parte della procedure può essere automatizzata ed in effetti non è necessario avere immediatamente presenti tutti i passi necessari per compilare ed installare il codice di un servizio di griglia.

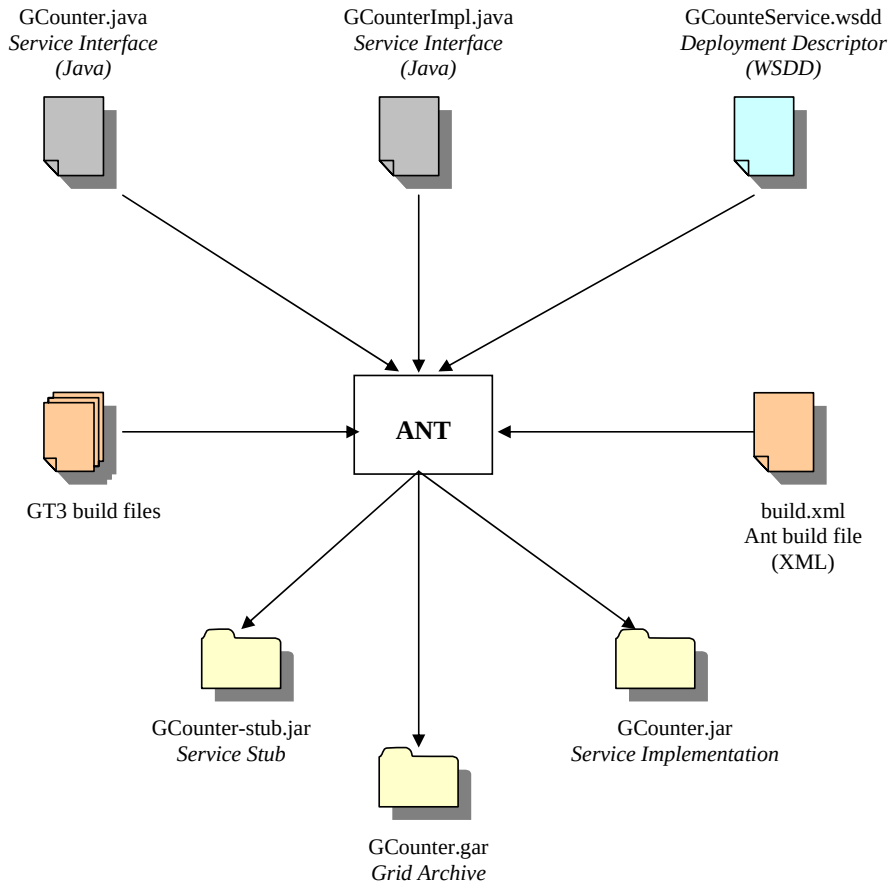
Nello sviluppo di un progetto, nella pratica, ci si limita a codificare un *build file*, l'equivalente di un *makefile* nel dialetto make, descrivendo i singoli passi necessari a generare il codice WSDL, gli stub files, gli archivi ed infine a produrre il file Grid Archive, che contiene tutti gli elementi necessari alla installazione ed all'effettivo utilizzo del servizio.

Nel nostro esempio lo sviluppatore scriverà tre sorgenti:

- GCounter.java l'interfaccia in codice Java
- GCounterImpl.java l'implementazione Java del servizio
- GCounterService.wsdd il deployment descriptor in linguaggio WSDD

e descriverà i passi necessari alla compilazione in un *build.xml* file (ancora una volta in un dialetto XML). La codifica di un build file risulterà ulteriormente semplificata dalla possibilità di fare riferimento ad una serie di *target* precodificati nel build file delle librerie OGSA di Globus Toolkit.

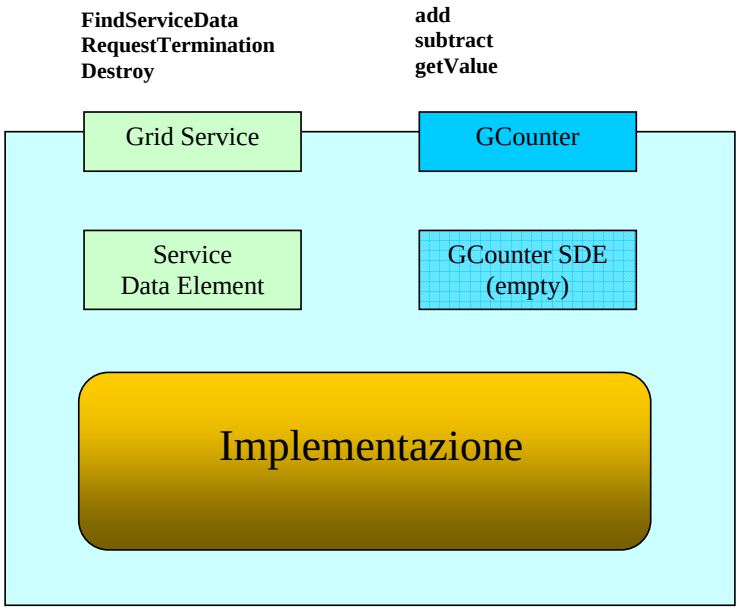
La compilazione, il deployment e l'undeployment di un servizio, richiederanno quindi poco più che l'esecuzione di un singolo comando ant.



Il Grid Archive file viene prodotto eseguendo il comando ant a fronte del build.xml file, viene installato con un comando del tipo "ant deploy –Dgar.name=GCounter.gar" e disintallato con un comando analogo "ant undeploy –Dgar.id=gvt".

Globus Tookit 3.0 permette di verificare il corretto funzionamento del servizio in un contenitore di servizi autonomo che può essere lanciato con un singolo comando: **globus-start-container**.

Il servizio di griglia GCounter offre, oltre all'interfaccia GridService comune a tutti i servizi di griglia, anche l'interfaccia definita dal GcounterPortType contenente le operazioni ***add***, ***subtract*** e ***getValue***. L'implementazione ne costituisce le funzionalità, il codice applicativo vero e proprio.



L'utente del servizio ancora una volta sarà una applicazione, il client, che verrà sviluppato con modalità del tutto analoghe all'implementazione del servizio. Analogamente all'ambiente Web Services la descrizione WSDL del servizio sarà accessibile via rete e sarà sufficiente a generare tutto il codice di interfaccia necessario nelle fasi di sviluppo e di test del client.

Per mantenere semplice il codice il nostro client si limiterà a chiamare il metodo ***add*** con il parametro 10, il metodo ***subtract*** con il parametro 3 ed il metodo ***getValue*** per ottenere il nuovo valore del contatore.

Al client verrà fornito il **Grid Service Handle** (GSH) del servizio, in pratica un URL, come prima parametro, direttamente dalla linea di comando.

Il meccanismo di base RPC è del tutto analogo a quello che abbiamo analizzato nel caso dei WebServices e permette di codificare il client in modo semplice ed elegante. Come è possibile verificare le modifiche sono minime e si limitano al codice necessario per interfacciarsi con le librerie OGSA di Globus Toolkit 3.

```
package gvt.sample.client;

import gvt.sample.GCounter.GCounterServiceLocator;
import gvt.sample.GCounter.GCounterPortType;
import java.net.URL;

public class GCounterClient
{
    public static void main(String[] args) throws Exception {
        // trasforma il parametro fornito dalla linea di comando
        // in un Grid Service Handle, un URL
        URL GSH = new java.net.URL ( args[0] );

        // Ottiene un handle del servizio
        GCounterServiceLocator counterService =
            new GCounterServiceLocator();
        GCounterPortType counter =
            counterService.getGCounterService ( GSH );

        // Chiama il metodo add con il parametro 10
        counter.add ( 10 );

        // Chiama il metodo subtract con il parametro 3
        counter.subtract ( 3 );

        // Chiama il metodo getValue per ottenere
        // il valore del counter
        int val = counter.getValue();

        // Stampa il risultato
        System.out.println ( val );
    }
}
```

Una volta compilato il codice Java del client in una classe, sarà possibile eseguirlo in una macchina virtuale Java con il comando:

```
java GcounterClient
   http://localhost:8080/gvt/sample/GcounterFactoryService/counter
```

a fronte della istanza *counter* del servizio di griglia GCounter.

Il Grid Service Handle, il GSH, del servizio è un **URI** (*Uniform Resource Identifier* RFC2396), nella pratica un URL della forma http://nomehost/nomedocumento, che non permette di comunicare con il servizio, ma solo di individuarlo in modo univoco sulla rete. Per poter comunicare con il servizio il GSH deve essere convertito in un **Grid Service Reference** (GSR).

Il **GSR** fornisce al client le informazioni necessarie per connettersi e dialogare con il servizio di griglia: protocolli, indirizzi, etc. e può scadere o perdere di validità. Può essere codificato come documento WSDL, ma implementazioni future potrebbero supportare altri ambienti come CORBA, DCOM, RMI. Questa distinzione aggiunge un livello di astrazione e lascia libertà nella scelta dei protocolli di comunicazione.

Nell'implementazione OGSA di Globus Toolkit 3.0 una operazione http GET su un GSH permette di ottenere un GSR valido. Nel nostro esempio GCounterClient accede al servizio GCounter trasformando il GSH, passato dalla linea di comando, in un oggetto della classe GcounterPortType attraverso i metodi della interfaccia **HandleResolver**:

```
// trasforma il parametro fornito dalla linea di comando
        // in un Grid Service Handle, un URL
        URL GSH = new java.net.URL ( args[0] );

// Ottiene un handle del servizio
        GCounterServiceLocator counterService =
            new GCounterServiceLocator();
        GCounterPortType counter =
            counterService.getGCounterService ( GSH );
```

Il GSH di un servizio di griglia è il riferimento, perpetuamente valido, all'istanza di un servizio, mentre il GSR viene creato su richiesta dell'applicazione e può scadere, perdere di validità: il servizio potrebbe essere *spostato* su un altro server, i protocolli e/o le implementazioni venire aggiornate, etc.

Nell'esempio analizzato siamo partiti dall'interfaccia del servizio codificata in Java e abbiamo quindi generato, in modo automatico, la descrizione WSDL della *service interface*. È sicuramente il modo più semplice di operare: una descrizione del servizio in Java è più *sintetica* della equivalente descrizione in WSDL (qualche decina di righe di codice contro qualche centinaio).

D'altra parte la service interface può essere codificata in **Grid WSDL** (GWSDL), un dialetto WSDL conforme alle specifiche OGSA, utilizzato da Globus Toolkit. Scrivere direttamente codice WSDL o GWSDL consente, a prezzo di una complessità leggermente più elevata, di ottenere una descrizione *semanticamente* più ricca ed un controllo maggiore sulla qualità del codice WSDL che descrive l'interfaccia del servizio. Per il resto, se si utilizza una service interface GWSDL, le procedure di compilazione e di deploying rimangono praticamente inalterate.

Seguendo passo la codifica dei portType GCounter in GWSDL, per prima cosa scriveremo l'elemento *radice* del documento GWSDL:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="GCounterService"
  targetNamespace="http://www.gvt.org/namespaces/0.1/samples/GCounter"
  xmlns:tns="http://www.gvt.org/namespaces/0.1/samples/GCounter"
  xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
  xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridWSDLExtensions"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
</definitions>
```

nel quale definiamo il nome del servizio ed il *namespace* XML che raggrupperà tutte le definizioni relativa alla interfaccia GCounter: un URI che non necessariamente è anche un URL valido.
L'elemento radice viene anche utilizzato per dichiarare tutti i namespaces che verranno utilizzati nella definizione del portType.

Il passo successivo è quello di importare, all'interno della <definitions>, il file OSGI GWSDL che

contiene una serie di definizioni che verranno utilizzate successivamente:

```
<import location="../../ogsi/ogsi.gwsdl"
  namespace="http://www.gridforum.org/namespaces/2003/03/OGSI"/>
```

Il passo più importante è la definizione, all'interno dell'elemento radice, del portType, utilizzando il tag <gwsdl:portType>, analogo al tag WSDL <portType>:

```
<gwsdl:portType name="GCounterPortType" extends="ogsi:GridService">
  <operation name="add">
    <input message="tns:AddInputMessage"/>
    <output message="tns:AddOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>

  <operation name="subtract">
    <input message="tns:SubtractInputMessage"/>
    <output message="tns:SubtractOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>

  <operation name="getValue">
    <input message="tns:GetValueInputMessage"/>
    <output message="tns:GetValueOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
</gwsdl:portType>
```

nel quale dichiariamo il nome del portType, GCounterPortType, ed il fatto che questo portType "estende" la definizione del portType ogsi:GridService. Questa possibilità, di estendere una definizione, ereditandone le caratteristiche, è una delle differenze più importanti tra GWSDL e WSDL. In effetti, in questo modo, dichiariamo che il GCounterPortType è *anche* un ogsi:GridService portType, ovvero che il nostro servizio è un servizio di griglia.

All'interno dell'elemento portType, in modo del tutto analogo ai documenti WSDL, dichiariamo le operazioni prescritte dall'interfaccia del servizio GCounter: **add**, **subtract** e **getValue**. Possiamo osservare che si tratta, in questo caso, di una procedura del tutto ripetitiva: le definizione differiscono solamente per il nome dei messaggi supportati.

Ciascun messaggio dovrà quindi essere definito separatamente, ma analizziamo, per comodità di esposizione, solo il caso dei messaggi relativi all'operazione **add**, cioè ***AddInputMessage*** ed ***AddOutputMessage***:

```
<message name="AddInputMessage">
  <part name="parameters" element="tns:add"/>
</message>

<message name="AddOutputMessage">
  <part name="parameters" element="tns:addResponse"/>
</message>
```

Ad ogni messaggio, con lo stesso nome con cui appare nei tag <input> ed <output>, è definito da un tag <part> che definisce i parametri del messaggio. La definizione di questi elementi **tns:add** e **tns:addResponse** farà parte del contenuto del tag <types>:

```
<types>
<xsd:schema targetNamespace="http://www.gvt.org/namespaces/0.1/samples/GCounter"
  attributeFormDefault="qualified"
  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="add">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="addResponse">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:int"/>
      </xsd:sequence>
     </xsd:complexType>
  </xsd:element>
</xsd:schema>

</types>
```

nel quale il parametro in input al messaggio **add** è definito come un intero XML ***xsd:int***, nello stesso modo in cui il parametro in output al messaggio **addResponse** è definito come un intero XML, rispettando il contratto stabilito per il metodo add della interfaccia del servizio.

Le restanti operazioni con i relativi messaggi vengono definite esattamente nello stesso modo: si tratta in pratica solo di eseguire la tediosa e ripetitiva procedure di sostituire ciò che deve essere sostituito.

In sintesi, il documento GWSDL che definisce l'interfaccia di un servizio di griglia, può essere codificato facilmente seguendo questa procedura:

1. definire l'elemento radice <definitions>
2. definire il <gwsdl:portType>
3. definire i <message> di input ed output del portType
4. definire i <types> per ogni elemento dei messaggi.

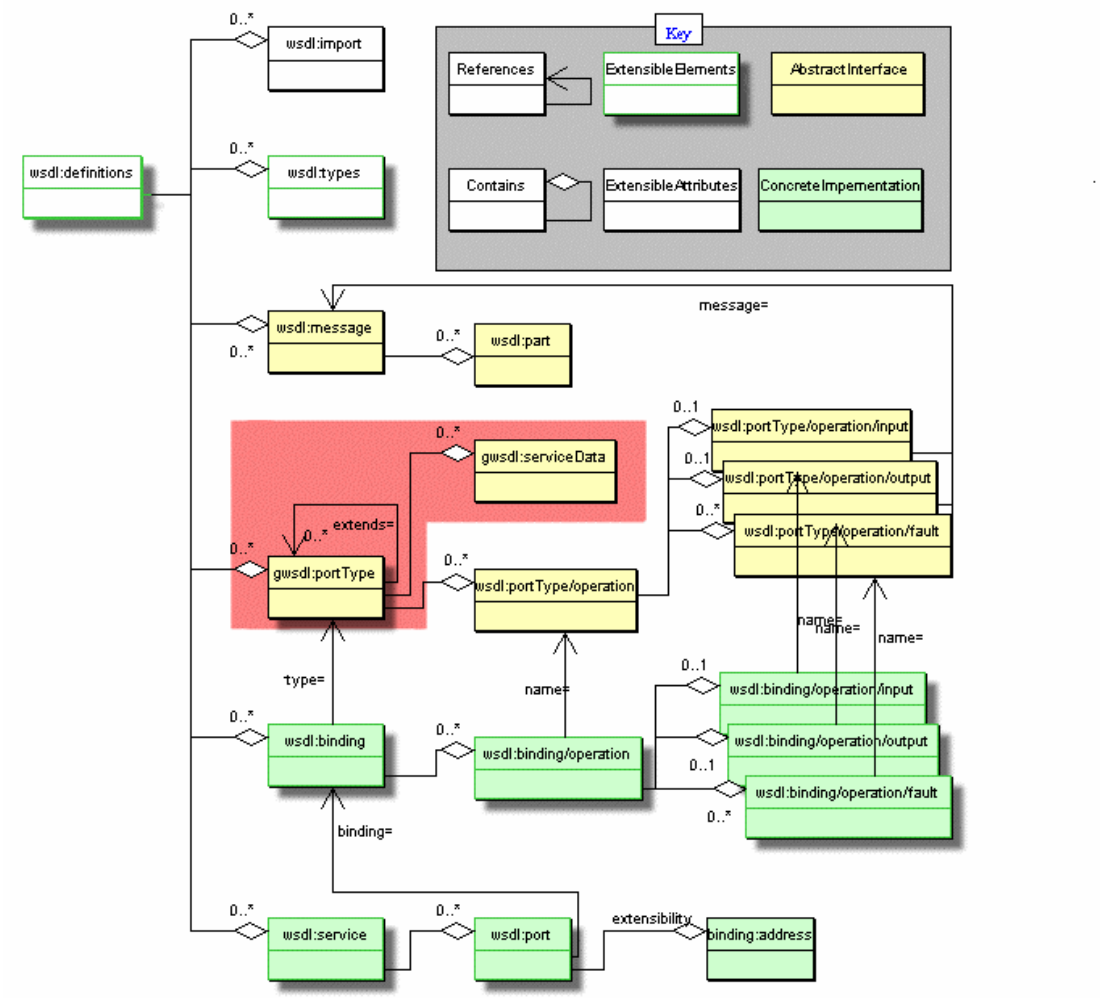
L'implementazione del server e del client procedono quindi, nel modo che abbiamo già analizzato, partendo dalla definizione GWSDL memorizzata nel documento GCounter.gwsdl invece che dall'interfaccia Java GCounter.java.

Il Grid Service WSDL estende la definizione dell'elemento portType del WSDL 1.1:

- il portType WSDL contiene solo operazioni, mentre un portType OGSI può contenere degli Elementi ServiceData
- nelle definizioni OGSI è possibile estendere la definizione di un portType.

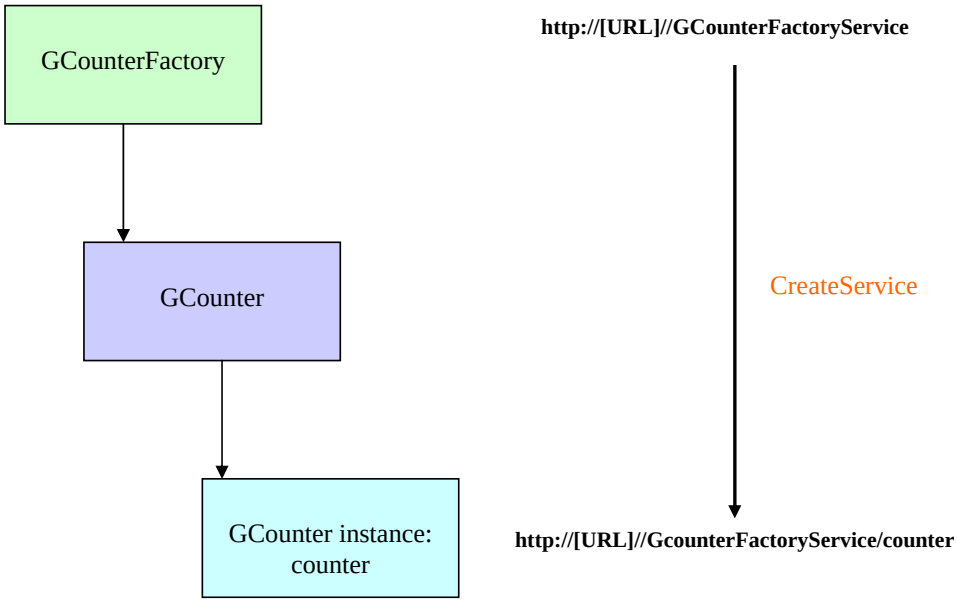
Le estensioni al linguaggio WSDL possono essere facilmente riconosciute dal prefisso *gwsdl* che le identifica come definizioni OGSI.

Il seguente schema, disponibile nell'articolo "Open Grid Service Infrastructure Primer", costituisce un utile riferimento per comprendere la struttura di un documento WSDL 1.1 e le estensioni GWSDL (evidenziate nel riquadro).



Poiché stiamo ereditando l'interfaccia ogsi:GridService al servizio GCounter sarà associata una interfaccia **Factory** in grado di creare nuove istanze del servizio GCounter, la GcounterFactoryService. Il metodo CreateService di questa interfaccia si occupa di creare nuove istanze del servizio, *istanziate* da una identica definizione.

L'operazione CreateService, dell'interfaccia Factory, si assicura che le istanze siano create in modo affidabile, una ed una sola volta e può essere estesa in modo da accettare parametri specifici di un servizio da passare alle istanze nel momento in cui vengono create. Assegna inoltre alle istanze un nome unico e ritorna il Gris Service Handle (GSH), l'URL, della istanza stessa.



Nello schema alla GcounterFactoryService, individuata attraverso il proprio GSH, viene richiesta la creazione di una istanza del servizio GCounter a cui viene assegnato un nome, ***counter***, ed un nuovo GSH associato a tale nome.

Dopo il deploy del servizio GCounter, sarà disponibile un nuovo servizio ***persistente***, la Factory, in grado di creare dinamicamente delle istanze ***transitorie*** del servizio di griglia GCounter. Tali servizi potranno essere terminati esplicitamente, utilizzando il metodo Destroy ereditato dalla definizione di GridService, o automaticamente assegnando ad essi un periodo di vita, un **lifetime**,

Una routine della libreria OGSA del Globus Toolkit o una semplice script possono quindi essere utilizzati per richiedere alla Factory di creare istanze del servizio:

ogsi-create-service http://localhost:8080/ogsa/services/gvt/samples/GCounterFactoryService ***counter***

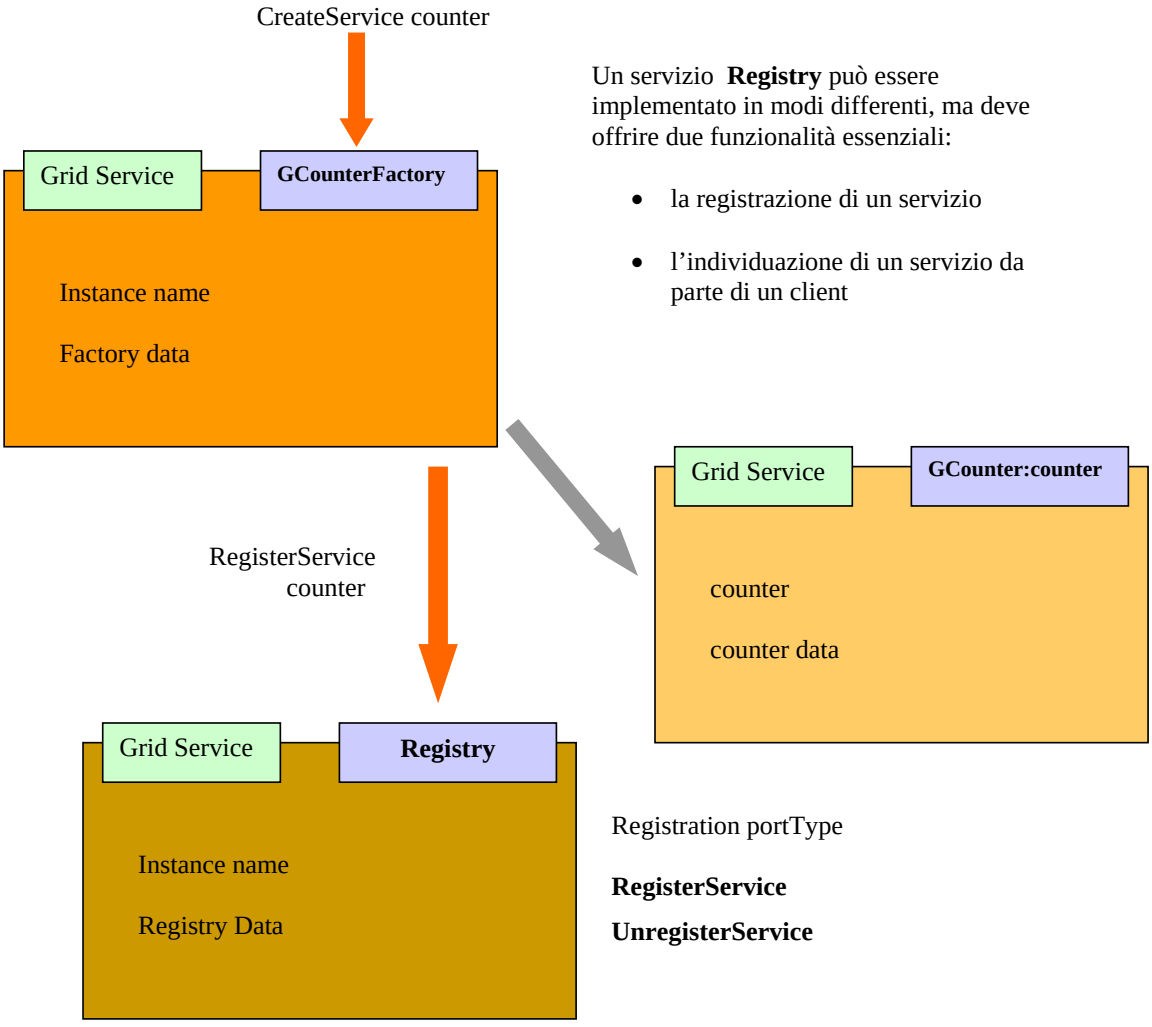
che ritornerà un service reference valido per l'istanza ***counter*** del servizio GCounter. L'istanza potrà quindi essere contattata al GSH

http://localhost:8080/ogsa/services/gvt/samples/GCounterFactoryService/***counter***

da una applicazione client.

L'architettura OGSA prevede una interfaccia **Registry** attraverso la quale le istanze dei servizi di griglia possono *pubblicare*, registrare la propria esistenza e le proprie caratteristiche in un catalogo, in un **Registry**: un servizio di griglia, funzionalmente analogo ad una Discovery Agency, nel quale possono essere effettuate ricerche, che mantiene una collezione di GSH.

Un Registry rende disponibili documenti WS-Inspection, definiti tramite il linguaggio WSIL (Web Services Inspection Language), un formato XML per accedere ad elenchi di Web Services, concepito da IBM e Microsoft.



Un servizio **Registry** può essere implementato in modi differenti, ma deve offrire due funzionalità essenziali:

- la registrazione di un servizio
- l'individuazione di un servizio da parte di un client

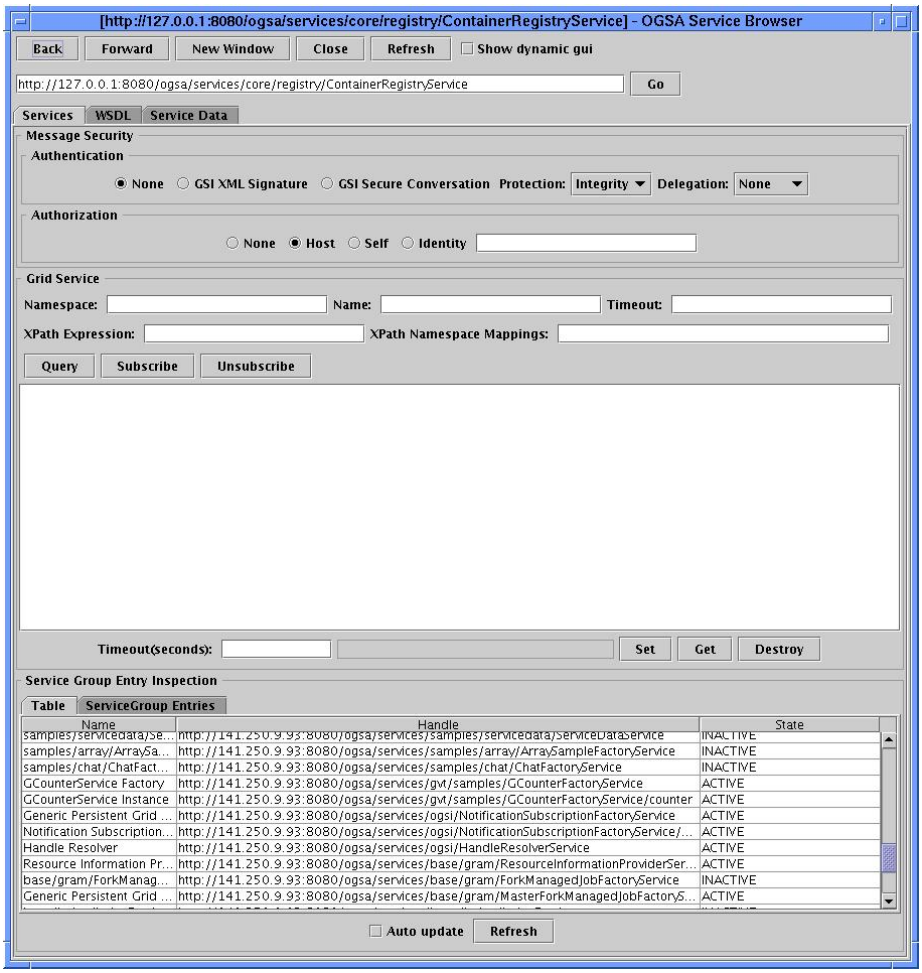
Registration portType

RegisterService

UnregisterService

Nella definizione OGSI i portTypes e le operazioni hanno una struttura differente: si rimanda al ***draft*** della Open Grid Service Infrastructure per una descrizione più completa e precisa.

Il registro di Globus Toolkit 3.0 può essere esplorato attraverso un ***Service Browser***: il browser viene attivato mediante il comando "**ant gui**" e contatta automaticamente il ContainerRegistryService locale.

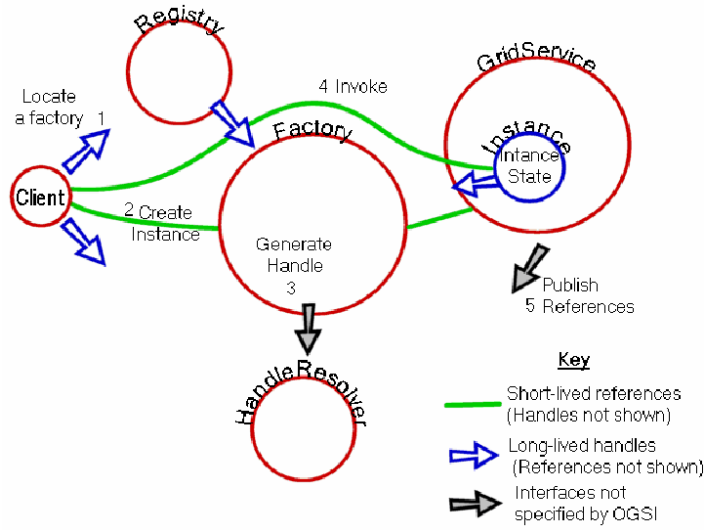


Mediante l'OGSA Service Browser è possibile esplorare l'elenco dei servizi disponibili, sia persistenti che transitori, contattare le factory per creare nuove istanze dei servizi, scegliere lo schema di autenticazione, visualizzare i service data elements, interagire con i servizi invocando i metodi disponibili.

È una prima implementazione di interfaccia grafica che permette di centralizzare ed automatizzare la gestione di una griglia di servizi. Il paradigma all'interno del quale ci stiamo muovendo è del tutto parallelo al modello di Ricerca/Pubblicazione/Interazione che abbiamo analizzato relativamente ai Web Services.

Uno schema, anche questa volta ripreso dall' OGSI Primer, permette di comprendere l'analogia esistente tra i due paradigmi:

dove, in termini di definizioni OGSI, l'interazione fra i vari elementi può essere descritta in questo modo:

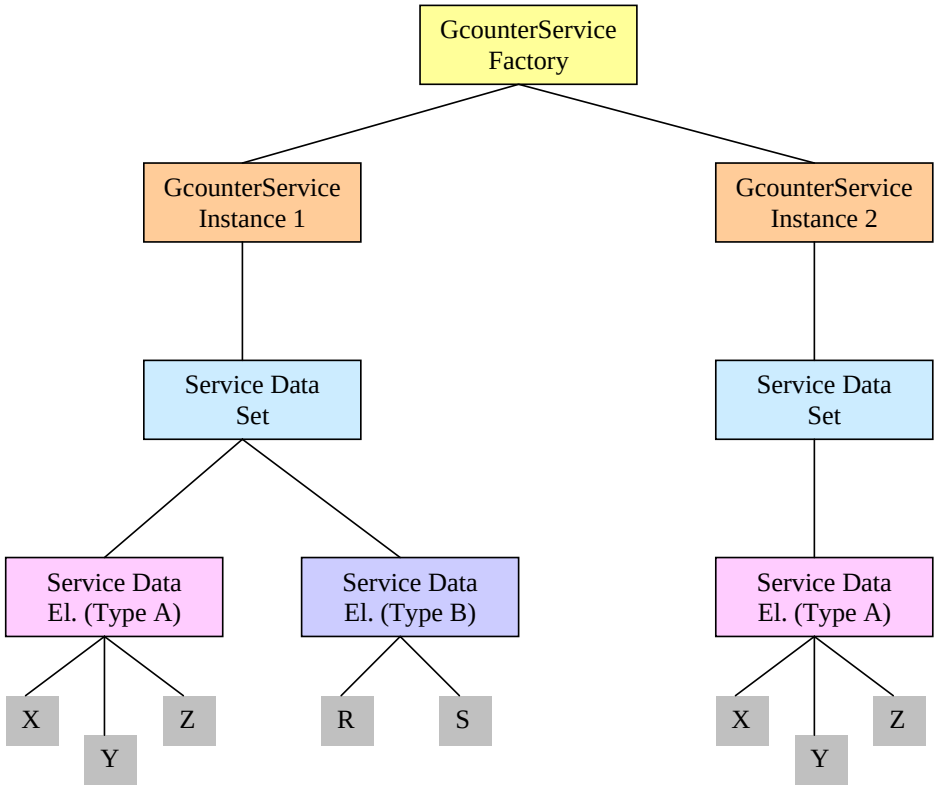


Il client contatta un *well-known* service, come il ContainerRegistryService, un ***Registry OGSA***, per identificare la Factory in grado di istanziare il servizio a cui è interessato. La Factory crea una nuova istanza del servizio e, interagendo con l'HandleResolver, assegna alla istanza un nuovo GSH univocamente determinato. Il GSH ed il GSR della istanza appena creata vengono quindi ritornati al client e pubblicati nel Registry.

In tal modo diviene possibile, interagendo con il Registry, accedere a tutti i servizi disponibili, sia persistenti (le Factory), sia transitori (le istanze).

Abbiamo già avuto modo di osservare che un servizio di griglia è un Web Service dotato di una interfaccia GridService e di un insieme di Service Data Element (SDE) che descrivono lo *stato* della istanza: si parla di *stateful service*. I Service Data sono un insieme strutturato di informazioni associate ad una istanza di un servizio di griglia e possono essere utilizzati per ottenere scopi differenti. Un Service Data può essere comune a tutte le istanze di un servizio e distribuire informazioni che permettono di scegliere un particolare servizio oppure può convogliare informazioni sullo stato di runtime di un servizio.

Riprendendo l'esempio del nostro GCounter, implementazioni differenti del GCounterPortype, possono essere associati a differenti Service Data Element:



Tutte le istanze di GCounter hanno un Service Data Set, che può contenere zero o più Service Data Elements (l'insieme può essere vuoto), ognuno contenente a sua volta un insieme differente di informazioni.

Nel nostro schema l'istanza 1 contiene due elementi differenti: il primo di tipo A, che contiene l'insieme di dati {X,Y,Z}, il secondo di tipo B che contiene l'insieme di dati {R,S}. In questo caso potrà essere *interrogata* relativamente all'insieme di informazioni {X,Y,Z,R,S}.

Sviluppando l'esempio relativo al GCounter possiamo aggiungere al contatore delle informazioni relative allo stato del servizio:

- l'ultima operazione effettuata **lastOp** una *stringa*
- il numero totale di operazioni effettuate **numOps** un *intero*

Tali informazioni, pure in un servizio così semplice, permetterebbero, se collezionate in una serie storica, di effettuare delle statistiche sul tipo e sul numero di richieste servite dal GCounter.

Stiamo andando a definire un Service Data Element, il cui nome sarà **GCounterData**, mediante uno schema XML:

```
<schema
targetNamespace="http://www.gvt.org/namespaces/0.1/samples/GCounterData"
        attributeFormDefault="qualified"
        elementFormDefault="qualified"
        xmlns="http://www.w3.org/2001/XMLSchema">

        <complexType name="GCounterDataType">
                <sequence>
                        <element name="lastOp" type="string"/>
                        <element name="numOps" type="int"/>
                </sequence>
        </complexType>

</schema>
```

nell'esempio un estratto dal file GcounterDataType.xsd.

Alla definizione GWSDL del servizio, GCounter.gwsdl, dovremo quindi apportare solo una limitata serie di modifiche, necessarie alla implementazione del Service Data Element GcounterData. Nella <definitions> dovremo includere il namespace relativo al GcounterData ed il namespace OGSI che contiene i tags legati ai service data:

```
<definitions name="GCounterService"
  targetNamespace="http://www.gvt.org/namespaces/0.1/samples/GCounter"
  xmlns:tns="http://www.gvt.org/namespaces/0.1/samples/GCounter"
  xmlns:data="http://www.gvt.org/namespaces/0.1/samples/GCounterData"
  xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
  xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridWSDLExtensions"
  xmlns:sd="http://www.gridforum.org/namespaces/2003/03/serviceData"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Dovremo quindi *importare* lo schema XML che definisce il GcounterDataType, ovvero il file GcounterDataType.xsd:

```
<import location="GCounterDataType.xsd"
  namespace="http://www.gvt.org/namespaces/0.1/samples/GCounterData"/>
```

ed infine aggiungere al tag <gwsdl:portType> la definizione delle proprietà dell'SDE:

```
<gwsdl:portType name="GCounterPortType" extends="ogsi:GridService">
...
<sd:serviceData name="GCounterData"
     type="data:MathDataType"
     minOccurs="1"
     maxOccurs="1"
     mutability="mutable"
     modifiable="false"
     nillable="false">
</sd:serviceData>
...
</gwsdl:portType>
```

mediante il tag <sd:serviceData>. In particolare abbiamo definito:

- il nome dell'SDE **GcounterData**
  deve essere unico nella definizione del servizio di griglia

- il tipo dell'SDE **data:MathDataType**
  il cui namespace ***data*** deve corrispondere alla definizione in GcounterDataType.xsd

Le altre proprietà richiedono un SDE che può assumere uno ed un solo valore, che può cambiare durante il periodo di vita del servizio, che non può essere NULL e non può essere modificato dal client.

Anche l'implementazione del servizio dovrà essere modificata allo scopo di fornire queste nuove funzionalità. La dichiarazione della classe sarà del tutto identica:

```
public class GCounterImpl extends GridServiceImpl
    implements GCounterPortType
```

mentre dovremo aggiungere due nuove variabili private:

```
    private ServiceData serviceData;
    private GCounterDataType gcounterData;
```

la prima serviceData è il Service Data Element, mentre la seconda gcounterData contiene i valore del data element. Dovremo inoltre aggiungere un nuovo metodo pubblico **postCreate**, che verrà chiamato automaticamente durante l'inizializzazione del servizio per impostare i valore iniziali del Service Data Element:

```
    public void postCreate ( GridContext context ) throws GridServiceException {
        serviceData = this.getServiceDataSet().create("GCounterData");

        gcounterData = new GCounterDataType();
        serviceData.setValue ( gcounterData );

        gcounterData.setLastOp ( "NONE" );
        gcounterData.setNumOps ( 0 );

        this.getServiceDataSet().add ( serviceData );
    }
```

Tutti i metodi dell'interfaccia **add**, **subtract** e **getValue**, corrispondenti alle operazioni del portType dovranno quindi prendersi carico di *aggiornare*, ogni volta che saranno richiamati, i valori di **lastOp** e **numOps**:

```
    public int add ( int val ) throws RemoteException {
        gcounterData.setLastOp ( "Add" );
        incrementOps();
        this.val = this.val + val;
        return this.val;
    }

    public int subtract ( int val ) throws RemoteException {
        gcounterData.setLastOp ( "Subtract" );
        incrementOps();
        this.val = this.val - val;
        return this.val;
    }

    public int getValue ( ) throws RemoteException {
        gcounterData.setLastOp ( "GetValue" );
        incrementOps();
        return this.val;
    }
```

In particolare il numOps verrà aggiornata richiamando un metodo privato (accessibile solo all'implementazione del servizio):

```
    private void incrementOps() {
        int numOps = gcounterData.getNumOps();
        gcounterData.setNumOps ( numOps + 1 );
    }
```

Qualche modifica al deployment descriptor, una veloce ricompilazione e … avremo una nuova implementazione del GCounter che potrà essere interrogata a runtime relativamente alle informazioni contenute nel SDE GCounterData.

Il tipo dati GCounterDataType, mediante il quale abbiamo definito i valori del service data GCounterData, è un Java Bean generato automaticamente, insieme agli stubs, dallo schema XML. Tale classe consente l'accesso ai membri lastOp e numOps mediante l'utilizzo di metodi *set* e *get* caratteristico del paradigma Java Beans.

L'operazione ***findServiceData*** del portType gridService permetterà ora ad un opportuno client di accedere in qualsiasi momento al Service Data Element GCounterData associato ad una istanza del servizio GCounter.

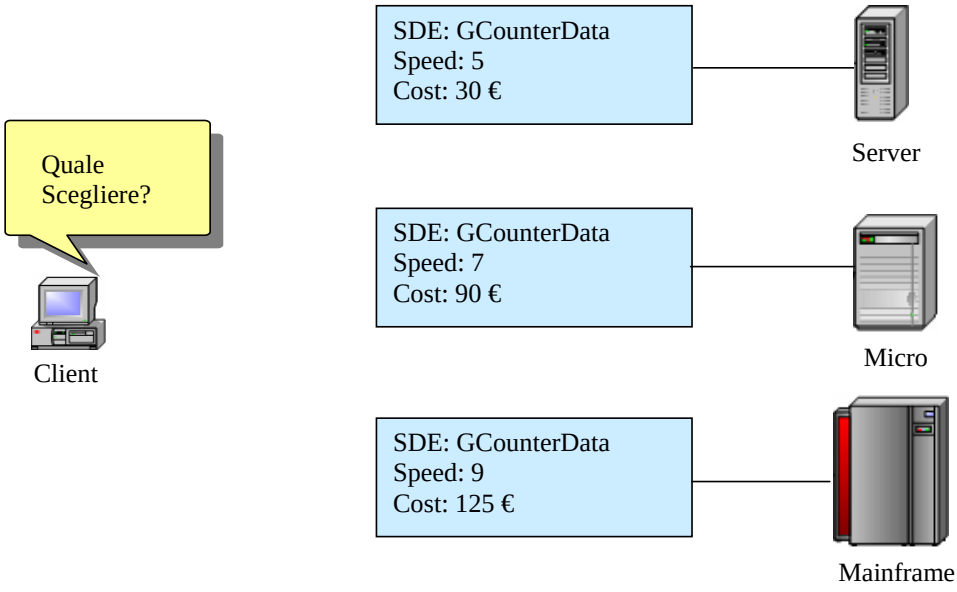
Il codice Java è relativamente semplice da scrivere: sono necessarie solo poche modifiche rispetto al client che abbiamo già analizzato. Si tratta di utilizzare una classe *helper* per risolvere il nome "GCounterData" in un oggetto Java di cui il client potrà effettuare il cast in un oggetto del Java Beans GCounterDataType.

```
package gvt.samples.client;

import org.gridforum.ogsi.OGSIServiceGridLocator;
import org.gridforum.ogsi.GridService;
import org.gridforum.ogsi.ExtensibilityType;
import org.gridforum.ogsi.ServiceDataValuesType;
import org.globus.ogsa.utils.AnyHelper;
import org.globus.ogsa.utils.QueryHelper;

import gvt.samples.wsdl.service.GCounterServiceGridLocator;
import gvt.samples.wsdl.GCounterPortType;
import gvt.samples.wsdl.servicedata.GCounterDataType;

import java.net.URL;

public class GCounterClient {
    public static void main ( String[] args ) throws Exception
    {
        // trasforma il parametro fornito dalla linea di comando
        // in un Grid Service Handle, un URL
        URL GSH = new java.net.URL (args[0] );

        // ottiene un reference ad un gridService portType
        OGSIServiceGridLocator locator = new OGSIServiceGridLocator();
        GridService gridService = locator.getGridServicePort ( GSH );

        // ottiene un Service Data Element "GCounterData"
        ExtensibilityType extensibility =
            gridService.findServiceData
            ( QueryHelper.getNamesQuery("GCounterData") );
        ServiceDataValuesType serviceData =
            AnyHelper.getAsServiceDataValues(extensibility);

        // cast del Service Data ad un oggetto della classe GCounterDataType
        GCounterDataType gcounterData =
            (GCounterDataType)
            AnyHelper.getAsSingleObject
            ( serviceData, GCounterDataType.class );

        // stampa i campi del Service Data Element
        System.out.println ( "ultima operazione: " + gcounterData.getLastOp());
        System.out.println ( "# di operazioni  : " + gcounterData.getNumOps());
    }
}
```

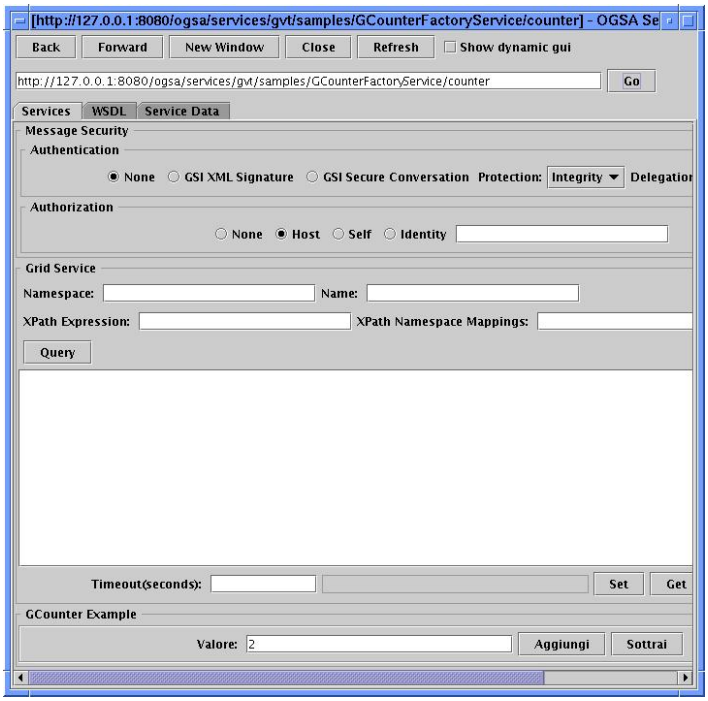
Questa tecnica, del tutto generale, può essere utilizzata per accedere, sia in lettura che in scrittura, ove possibile, un service data element per analizzare il contenuto dei meta-data o degli state-data.

Sulla griglia computazionale un servizio di catalogazione dei servizi disponibili potrebbe utilizzare i Service Data Element per soddisfare una particolare esigenza di un client: possiamo immaginare la nostra applicazione richiedere un servizio su una macchina con particolari caratteristiche o con una particolare implementazione del GCounter.

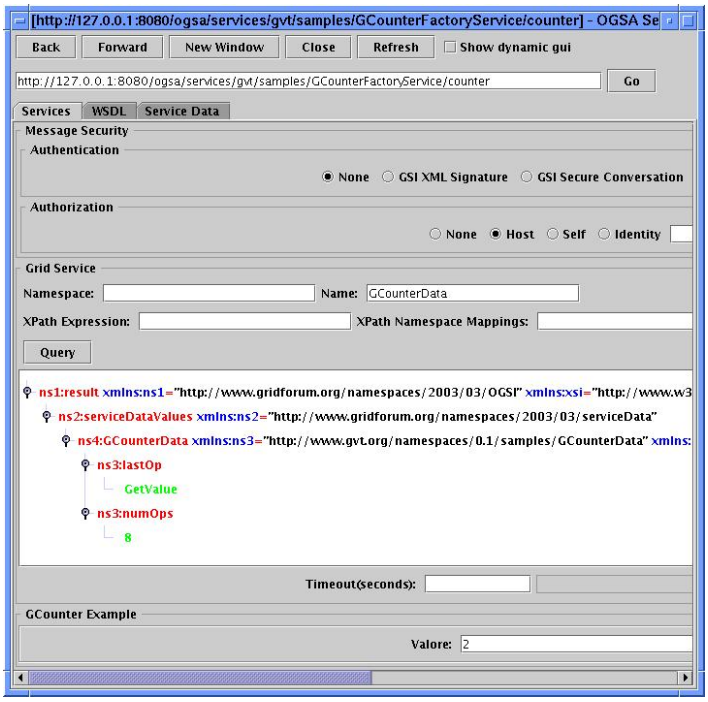
In un esempio più concreto la richiesta potrebbe vertere intorno alle perfomances della macchina o al costo di utilizzo: in questo caso al contatore sarebbero associati dei meta-data memorizzati in un opportuno SDE.



È possibile interagire con il servizio e con i suoi data elements mediante il Service browser:



Dal pannello principale del browser è possibile creare l'istanza *counter* o selezionarla nell'indice se esiste già nel catalogo del Registry e quindi visualizzare la finestra che permette di interagire con il servizio. Nell'esempio il pannello "GCounter Example", in basso, permette di visualizzare il valore del contatore (richiamando il metodo getValue) ed invocare i metodi ***add*** mediante il bottone "**Aggiungi**" e ***subtract*** mediante il bottone "**Sottrai**", passando il valore inserito nel campo "**Valore**". Il pannello stesso non è altro che una sottoclasse del controllo Swing JPanel, opportunamente configurata al Service Browser, in possesso di un GSR valido del servizio GCounter il cui GSH è visualizzato nel campo accanto al bottone "**Go**" del browser.



Dal browser si può interagire con il Service Data Element GCounterData dell'esempio, inserendo il nome del Service Data Element nel campo "**Name**", del pannello "GridService", e agendo sul bottone "**Query**" dello stesso pannello.

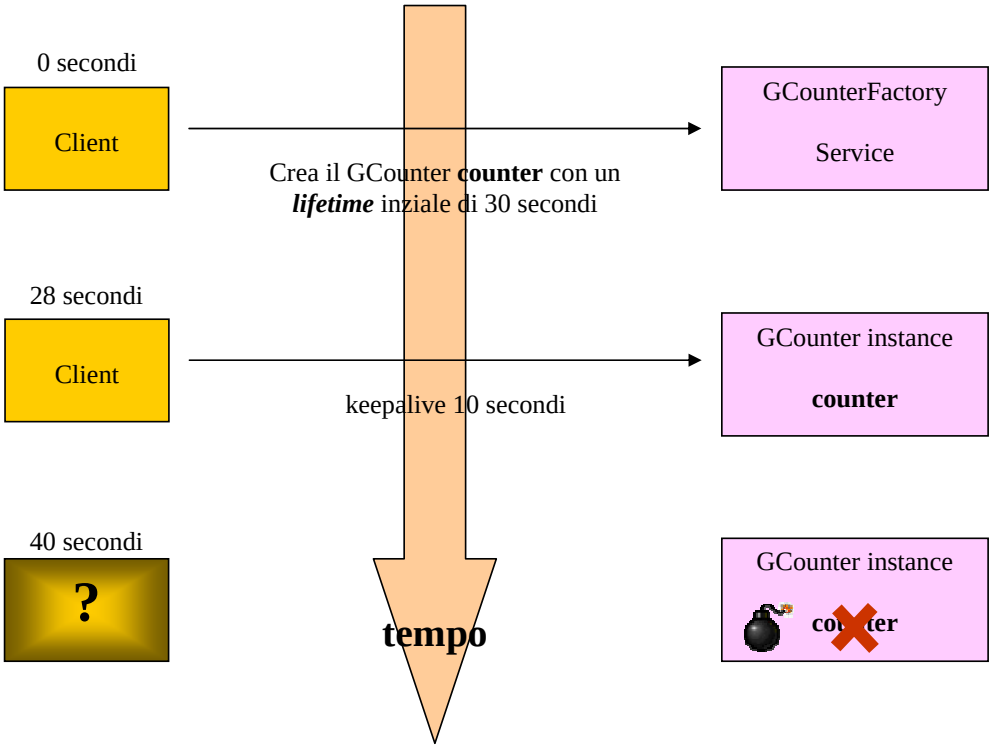
La ***sicurezza*** è un aspetto molto importante di qualunque ambiente distribuito: in Globus Toolkit queste problematiche vengono affrontate (anche se, a mio avviso, non completamente risolte) nella **Grid Security Infrastructure** (GSI). Il problema della sicurezza sulla Griglia Computazionale va ben oltre gli scopi di questa relazione; descriveremo solo brevemente le tecnologie sulle quali è fondata l'infrastruttura GSI.

Nei pannelli del service browser possiamo notare che sono previste diversi livelli di autenticazione ed autorizzazione all'uso del servizio, tra i quali lo scambio **sicuro** di informazioni attraverso le librerie GSI.

La Grid Security Infrastructure, già presente in Globus Toolkit 2.0, è una tecnologia basata su un sistema centralizzato di certificazione X.509: la mutua autenticazione è fondata sull'esistenza di Certification Authorities che *firmano*, in modo sicuro, dei certificati. Le informazioni scambiate vengono *cifrate* utilizzando la tecnologia ***Secure Socket Layer*** (SSL) ed un meccanismo a doppia chiave pubblica e privata (**RSA**), fondato sul principio che sia *impossibile* fattorizzare un numero intero molto grande (dell'ordine delle migliaia di cifre binarie). La GSI prevede un meccanismo di delega che semplifica le attività dell'utente finale: con una singola operazione di autenticazione, "*single sign-on*", è possibile utilizzare tutti i servizi di griglia per i quali si è autorizzati.

I Grid Services possono essere distrutti *esplicitamente* o *automaticamente*: l'operazione ***Destroy*** dell'interfaccia GridService è utilizzato per richiedere la distruzione *esplicita* di un servizio. Le operazioni di ***RequestTermination*** permettono invece di ottenere e modificare l'istante nel quale il servizio sarà terminato in modo *automatico*.

Lo scopo dell'operazione ***Destroy*** è di per se abbastanza evidente. I meccanismi di terminazione automatica sono stati implementati per evitare situazioni nelle quali un malfunzionamento da parte del client o della rete potrebbero *consumare* inutilmente le risorse del sistema ospite.



In un esempio più evidente il servizio potrebbe allocare, durante l'esecuzione, gigabytes di spazio disco o di memoria centrale, oppure impegnare una risorsa fisica *importante*: se il contatto con il l'applicazione dovesse venire meno per una qualunque ragione (caduta del client, della connessione di rete, etc.), le risorse sarebbero comunque rilasciate dopo un periodo di tempo noto a priori.

Un gruppo di interfacce *opzionali* particolarmente interessanti sono legate al paradigma osservatore/osservabile, noto negli ambienti grafici avanzati con il nome di Model-View-Controller.

Le interfacce di **Notifica** permettono l'invio di messaggi *asincroni* da un **NotificationSource** ad un insieme di **NotificationSink**:

- una istanza di un servizio di griglia che implementa l'interfaccia **NotificationSource** è in grado di spedire messaggi asincroni, *notifiche*, ai servizi che ne abbiano fatto richiesta, che si siano *abbonati* alla ricezione di una certa classe di messaggi;

- l'operazione **DeliverNotification** del portType **NotificationSink** viene implementata dai servizi che ricevono messaggi di notifica;

- NotificationSource e NotificationSink si scambiano messaggi di notifica, eventi asincroni, in un formato XML che è determinato dal tipo di abbonamento richiesto;

- l'abbonamento ad una particolare classe di messaggi viene richiesto mediante una *Subscription Expression*, sempre in formato XML.

L'esistenza di queste interfacce è motivata dalla necessità di comunicare un cambiamento nello stato di un servizio di grglia (osservabile) ad un insieme, potenzialmente molto grande, di applicazioni (osservatori). La soluzione più semplice potrebbe sembrare, ad un primo approccio, delegare tale funzionalità alle applicazioni: il client, periodicamente, invia al servente un richiesta di aggiornamento dei dati relativi allo stato del servizio. Questa soluzione, applicata ad una griglia distribuita di servizi, presenta immediatamente dei problemi di efficienza praticamente impossibili da risolvere: il traffico di rete tende a crescere in modo incontrollabile mano a mano che cresce il numero di servizi coinvolti.

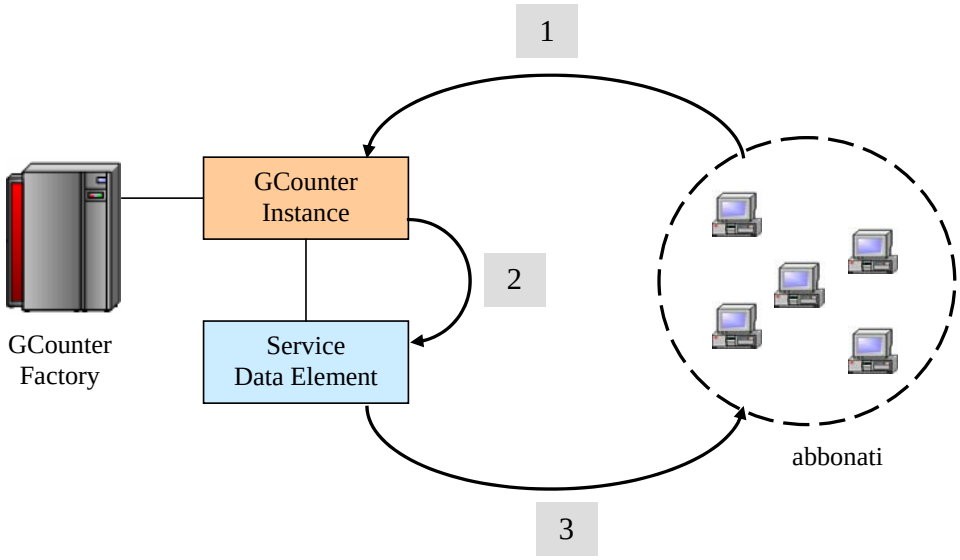
Nel paradigma osservatore/osservabile, *l'osservatore* (un altro servizio, non necessariamente un client), richiede al server, *l'osservabile*, di essere informato relativamente all'evento di un cambiamento di stato del servizio: questo passo viene definito *abbonamento* o *registrazione* dell'osservatore.
Il server mantiene un elenco di tutti gli osservatori che si sono registrati e degli stati osservabili a cui sono interessati e nel momento (e solo in quello) in cui si verifica uno degli eventi richiesti, comunica con tutti (e soli) gli osservatori abbonati a tale evento.

Questo approccio è decisamente più efficiente del precedente, anche se più complesso da implementare ed utilizzare. Esistono due modi di applicare il modello Osservatore/Osservabile:

- nel modello **Pull** l'osservabile notifica all'osservatore solamente che un determinato tipo di evento, di cambiamento di stato è avvenuto, non invia all'applicazione registrata alcuna informazione aggiuntiva; è responsabilità dell'applicazione inviare un ulteriore richiesta nella quale sono specificate le informazioni alle quali è interessata;

- nel modello **Push** i dati relativi al verificarsi di un evento relativo ad particolare osservabile vengono notificati direttamente all'applicazione, non è necessaria alcuna ulteriore forma di comunicazione tra osservatore ed osservabile.

In Globus Tookit 3.0 la notifica di eventi è strettamente correlata con i Service Data Element. L'osservatore si registra, si abbona, per essere informato di cambiamenti di stato che possano verificarsi in un particolare SDE di una istanza di un servizio di griglia.



1. **addListener**: il client si registra, si abbona, a ricevere le notifiche di eventi relativi ad un Service Data Eelement, specificato nella chiamata;

2. **notifyChange**: ogni qualvolta si verifica un cambiamento, il Service Data Element notifica l'evento a tutti i client che si sono registrati;

3. **deliverNotification**: i client registrati "devono" implementare l'interfaccia Java ***NotificationSinkCallback*** e fornire una implementazione di questo metodo che verrà richiamato tutte le volte che il Service Data Element invia una notifica.

Qualche frammento di codice Java può essere utile per visualizzare come queste primitive vengono effettivamente utilizzate:

```
    public int add ( int val ) throws RemoteException {
        gcounterData.setValue ( gcounterData.getValue() + val );
        gcounterData.setLastOp ( "Add" );
        incrementOps();
        serviceData.notifyChange();

        return gcounterData.getValue();
    }
```

in questo caso una istanza del servizio GCounter chiama il metodo **notifyChange** dell'SDE gcounter per notificare ai client registrati un cambiamento di stato relativo al metodo add.

Le applicazioni interessate a tale evento si sono preventivamente registrate utilizzando il metodo **addListener** di un ***NotificationSinkManager***:

```
        NotificationSinkManager notifManager =
            NotificationSinkManager.getManager();
        notifManager.startListening ( NotificationSinkManager.MAIN_THREAD );
        String sink =
            notifManager.addListener("GCounterData", null, GSH, this);
```

ed avranno implementato il metodo **deliverNotification**:

```
   public void deliverNotification ( ExtensibilityType any )
        throws RemoteException {
. . .
            ServiceDataValuesType serviceData =
                AnyHelper.getAsServiceDataValues(any);
            GCounterDataType gcounterData = (GCounterDataType)
                AnyHelper.getAsSingleObject
                  (serviceData, GCounterDataType.class);
. . .
    }
```

della interfaccia ***NotificationSinkCallback***, la cui implementazione è obbligatoria per le classi che vengono registrate per ricevere notifiche da un NotificationSource. Il frammento utilizzato rende evidente che l'applicazione riceve, insieme alla notifica, una copia del Service Data Element che ha generato l'evento.

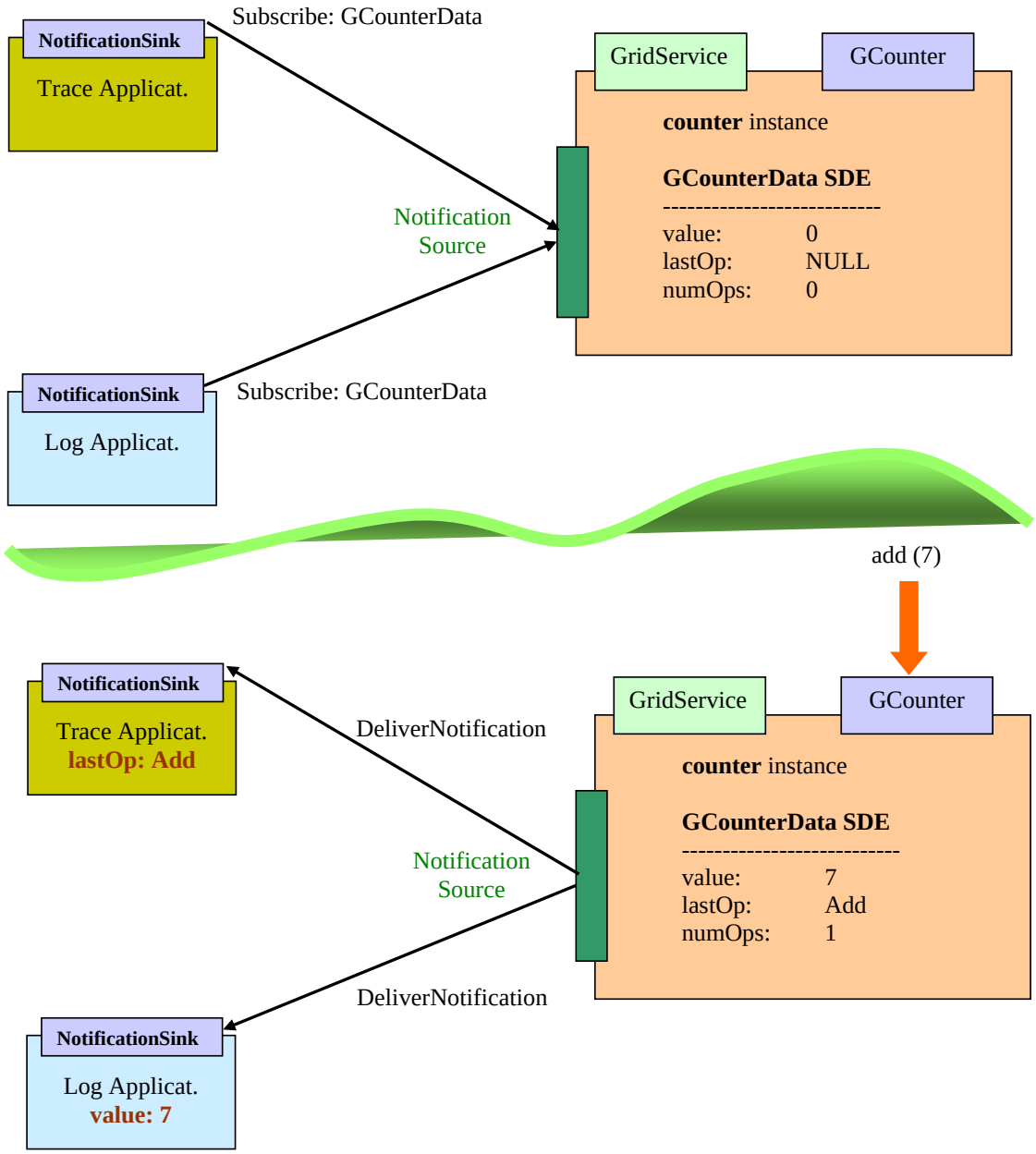
Nella implementazione Globus Toolkit 3.0 del paradigma, gli osservabili sono definiti *notification sources*, mentre gli osservatori vengono chiamati *notification sinks*.

Tornando al nostro esempio del contatore possiamo tentare di analizzare come differenti applicazioni potrebbero utilizzare il Service Data Element GcounterData, al quale è stato aggiunto l'elemento **value**

```
<schema
targetNamespace="http://www.gvt.org/namespaces/0.1/samples/GCounterData"
        attributeFormDefault="qualified"
        elementFormDefault="qualified"
        xmlns="http://www.w3.org/2001/XMLSchema">

        <complexType name="GCounterDataType">
                <sequence>
                        <element name="value"  type="int"/>
                        <element name="lastOp" type="string"/>
                        <element name="numOps" type="int"/>
                </sequence>
        </complexType>

</schema
```

per monitorare lo stato del servizio.

Nel nostro esempio due applicazioni differenti, una di traccia ed una di logging, si registrano al Service Data Element GCounterData con scopi differenti. La ***Trace Application*** manterrà una traccia di tutte le operazioni eseguite dal servizio, mentre la ***Log Application*** registrerà i valori assunti dal contatore. Possiamo immaginare che entrambi client registrino, insieme alle informazioni fornite dal SDE, l'istante temporale nel quale gli eventi si sono verificati.



10. OGSA: Conclusioni

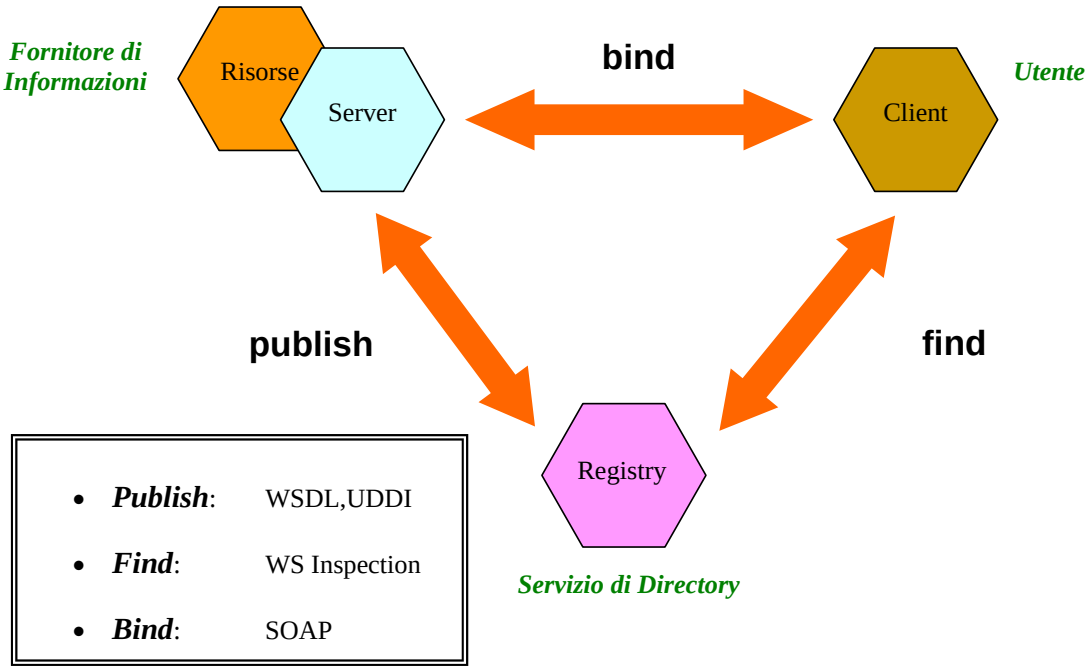
La Open Grid Service Archtecture è una architettura orientata ai servizi:

- dai Web Services
  - il paradigma RPC
  - la definizione di interfacce come documenti XML
  - protocolli multipli

- dal Grid
  - affidabilità
  - sicurezza
  - interfacce comuni per
    - l'individuazione dei servizi
    - la gestione del ciclo di vita
    - la notifica di eventi asincroni
  - la virtualizzazione delle risorse

- implementata su sistemi ospiti multipli:
  - "C", J2EE, .NET, etc.

Il modello astratto definito nella OGSA è molto simile al modello proposto dai Web Services e sotto molto punti di vista compatibile e potenzialmente interoperante.



È naturale chiedersi se il disegno dei Grid Services sia Object Oriented. Sotto molti punti di vista lo si può ritenere tale: le estensioni GWSDL prevedono la ereditarietà e i servizi sono dotati di uno stato interno accessibili solo attraverso operazioni ben definite. Ma il modello intorno al quale sono disegnati i servizi di griglia:

- mette l'accento su *incapsulazione* piuttosto che *ereditarietà*;
- non richiede una ***implementazione*** Object Oriented;
- è focalizzato sui documenti.

Il disegno Web/Grid Services può essere utilizzato con estrema facilità per incapsulare delle applicazioni *legacy* all'interno dei servizi, definendo le interfacce opportune. In tal modo applicazioni già esistenti, spesso complesse ed estremamente costose, possono essere convertite in servizi di griglia.

11. Pubblica Amministrazione

Il termine *e-government* è entrato a far parte del lessico della Pubblica Amministrazione italiana ad indicare il riconoscimento dell'impatto e della innovazione tecnologica nella organizzazione delle pubbliche amministrazioni. Dall'esperienza spontanea delle reti civiche ai recenti piani locali e nazionali di attuazione, un lungo percorso di maturazione giuridica e tecnologica è stato compiuto.

La Pubblica Amministrazione, soprattutto le amministrazioni locali, hanno iniziato ad affacciarsi ad Internet a partire dalla metà degli anni '90 sotto la spinta, nella maggior parte dei casi, di iniziative isolate, nate senza un quadro di coordinamento generale. Inizialmente la motivazione principale che ha mosso lo sviluppo di una infrastruttura tecnologica è stata la necessità di essere presenti sulla rete, tradotta nella implementazione di un sito con funzioni meramente conoscitive, una "proiezione esterna" dell'organizzazione interna dell'ente pubblico.

I problemi tecnologici affrontati dagli enti, durante questa fase dello sviluppo, sono stati strettamente legati all'evoluzione delle infrastrutture telematiche del nostro paese e possono essere sintetizzati nella necessità di connettersi alla rete globale in modo adeguato, utilizzando software applicativo di base, largamente disponibile. D'altra parte il grande successo dell'esperienza delle reti civiche ha contribuito in modo significativo alla estensione dei servizi a fasce sempre più ampie della collettività e ad un aumento della complessità applicativa.

Da siti Web con funzionalità conoscitive ed informative, spesso semplici collezioni di pagine HTML, di moduli e di documenti, l'attenzione e l'interesse si sono spostati verso siti in grado di fornire veri e propri servizi applicativi, in qualche caso accompagnati da vere e proprie transazioni di natura finanziaria.

L'interesse della Pubblica Amministrazione si è quindi focalizzato sulla necessità di coordinare tali iniziative in una rete complessa in grado di cooperare per fornire ai soggetti interessati, cittadini, enti, imprese, un sistema integrato di servizi.

La stessa natura degli enti pubblici rende particolarmente complesso e gravoso un tale processo di coordinazione: si tratta, nella maggior parte dei casi, di organizzazioni indipendenti che utilizzano internamente differenti modelli organizzativi, utilizzando paradigmi applicativi, ambienti di sviluppo, architetture e protocolli sostanzialmente eterogenei.

Gli enti hanno infatti un'ampia autonomia organizzativa e si presentano sulla rete come entità autonome che implementano funzionalità applicative che fanno uso di tecnologie potenzialmente *incompatibili*, ma l'esigenza di fornire servizi attraverso l'interfaccia Web rappresenta di per sè un potenziale fattore unificante.

È una opportunità che è stata colta dalle strutture di governo del nostro paese nelle iniziative di *e-government* rivolte alla riorganizzazione della Pubblica Amministrazione, sia al livello centrale che al livello di enti territoriali.

12. La Rete Nazionale

Al livello tecnologico il cardine intorno al quale ruotano le iniziative rivolte a coordinare e sviluppare i servizi informatici e telematici della Pubblica Amministrazione è l'infrastruttura definita nelle caratteristiche dalla **Rete Nazionale** dal Dipartimento per l'Innovazione e le Tecnologie della Presidenza del Consiglio dei Ministri.

La Rete Nazionale si pone come obiettivo la realizzazione di una infrastruttura per l'interconnessione di tutte le pubbliche amministrazioni, caratterizzata da livelli di servizio omogenei su tutto il territorio nazionale e pone come vincolo architetturale l'adozione del protocollo IP come tecnologia trasmissiva unificante.

La Rete Nazionale, a regime, dovrebbe fornire agli utenti un servizio di interconnessione che consenta comunicazioni sicure ed interoperabilità applicativa tra i domini informativi delle diverse amministrazioni.

La connessione alla Rete Nazionale avverrà:

- per le amministrazioni centrali mediante il collegamento alla Rete Unitaria della Pubblica Amministrazione (**RUPA**);

- per le amministrazioni locali attraverso la RUPA, le **RUPAR** (reti territoriali) o un *Internet Service Provider* (**ISP**) che soddisfi le prescrizioni tecniche del Dipartimento per l'Innovazione e le Tecnologie.

Il modello di internetworking proposto nel quadro architetturale e tecnologico del programma di e-government si ispira ai seguenti principi:

- la Rete Nazionale si pone l'obiettivo di permettere l'interconnessione, *any to any*, tra tutti i soggetti rappresentati dalle amministrazioni dello stato, dalle regioni e dagli enti locali, sfruttando pienamente le potenzialità offerte dallo sviluppo e dalla diffusione delle tecnologie;

- la Rete Nazionale completa un quadro che vede da un lato il consolidamento della RUPA e dall'altro lo sviluppo ed il consolidamento delle reti regionali, territoriali o RUPAR, garantendo l'interoperabilità ed estendendo la possibilità di accesso e di cooperazione attraverso l'offerta di mercato costituita dai servizi di ISP qualificati;

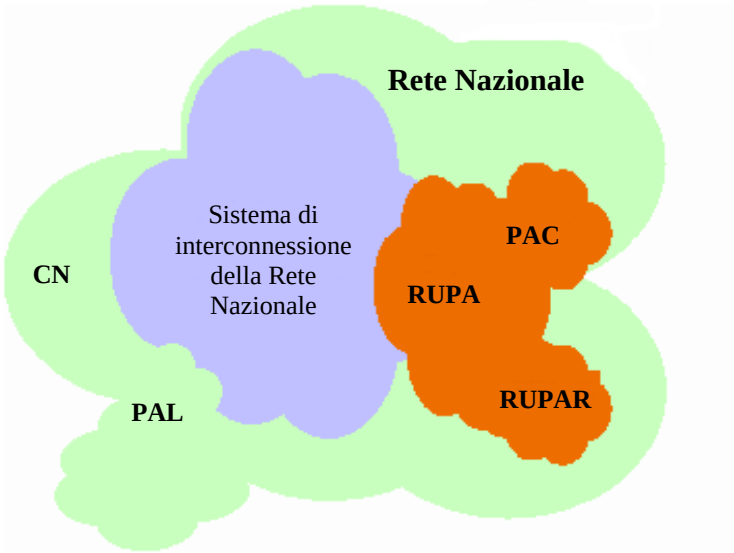
- la Rete Nazionale è concepita per una utilizzazione rapida ed ottimale delle opportunità offerte dal mercato delle telecomunicazioni e per generare una domanda qualificata agli operatori del settore, capace di potenziare e far evolvere su livelli qualitativamente migliori l'infrastruttura Internet del paese.

La Rete Nazionale si configura quindi come una *internet* di reti ***paritetiche***, interoperanti mediante la *TCP/IP protocol suite*, che dovrà interconnettere i diversi soggetti della Pubblica Amministrazione

locale e centrale, aderendo, almeno in parte, al paradigma sul quale si è fondata, più in generale, l'evoluzione di Internet.

I servizi di interconnessione della Rete Nazionale saranno quindi forniti, oltre che dalla RUPA, dalle Reti regionali, dalle *Community Network* (CN) e dalle altre reti di settore e di categoria. Inoltre i soggetti coinvolti potranno stipulare contratti con ISP che avranno stipulato a loro volta un contratto con gli *Exchange Point Operator* (EPO) che garantiscono l'interconnessione delle amministrazione locali e centrali.

L'affidabilità del sistema di interconnessione dovrebbe essere garantita dai requisiti richiesti ai provider della Rete Nazionale



Un semplice schema può essere di aiuto a rappresentare la Rete Nazionale comprendente le amministrazioni centrali (**PAC**), quelle locali (**PAC**) e le *Community Network* (**CN**).

Secondo queste definizioni la Rete Nazionale si prefigura come una rete che assume gli standard e i protocolli di comunicazione di Internet a fondamento della infrastruttura che dovrà fornire agli enti ed ai soggetti interessati il *trasporto* delle informazioni. D'altra parte uno sforzo progettuale notevole dovrà essere dedicato ad una organizzazione in grado di garantire elevati livelli di sicurezza e di qualità dei servizi.

Eliminato: tes

L'infrastruttura delineata nel quadro architetturale ben si presterebbe ad una applicazione delle tecnologie più avanzate sviluppate per affrontare problematiche di *Business to Business* (B2B) o di applicazioni complesse nel campo dell'*High Performance Computing* (HPC).

13. E-Government: il modello

La riorganizzazione informatica e telematica della Pubblica Amministrazione ha come obiettivo fornire all'utente, al cittadino, all'impresa servizi moderni con i quali sia facile operare. Una Pubblica Amministrazione efficiente e trasparente nei suoi compiti e nel suo grande patrimonio informativo potrebbe divenire un fattore di innovazione e di competitività per il paese.

Un sistema di ***e-government*** rappresenterà anche un potente strumento di coinvolgimento e partecipazione dei cittadini ai processi decisionali, evolvendo verso modelli innovativi di democrazia.

Il modello è composto da sei elementi chiave:

1. **Erogazione servizi**
   Un insieme di servizi dovranno essere resi disponibili attraverso modalità innovative ed ad un livello di qualità elevato a tutti i soggetti coinvolti: tali servizi saranno forniti in modo da mascherare la complessità interna e con un unico punto di accesso, anche se coinvolgono più amministrazioni.

2. **Riconoscimento digitale**
   Riconoscimento dell'utente e di *firme sicure* attraverso la Carta di Identità Elettronica, la Carta Nazionale dei Servizi e la firma digitale.

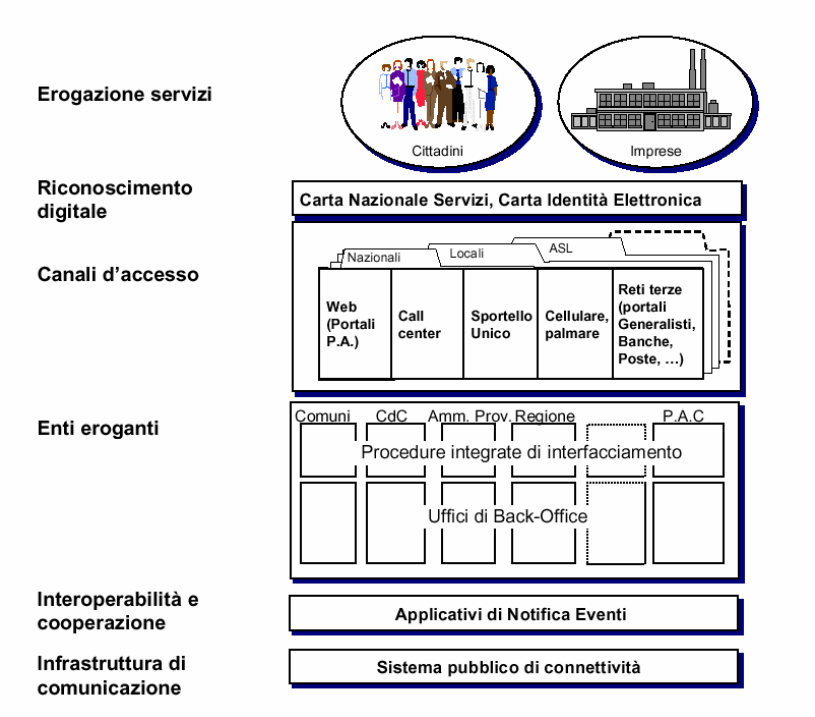
3. **Canali di Accesso**
   Canali innovativi: Internet, Call Center. Cellulare, reti di terzi, etc.

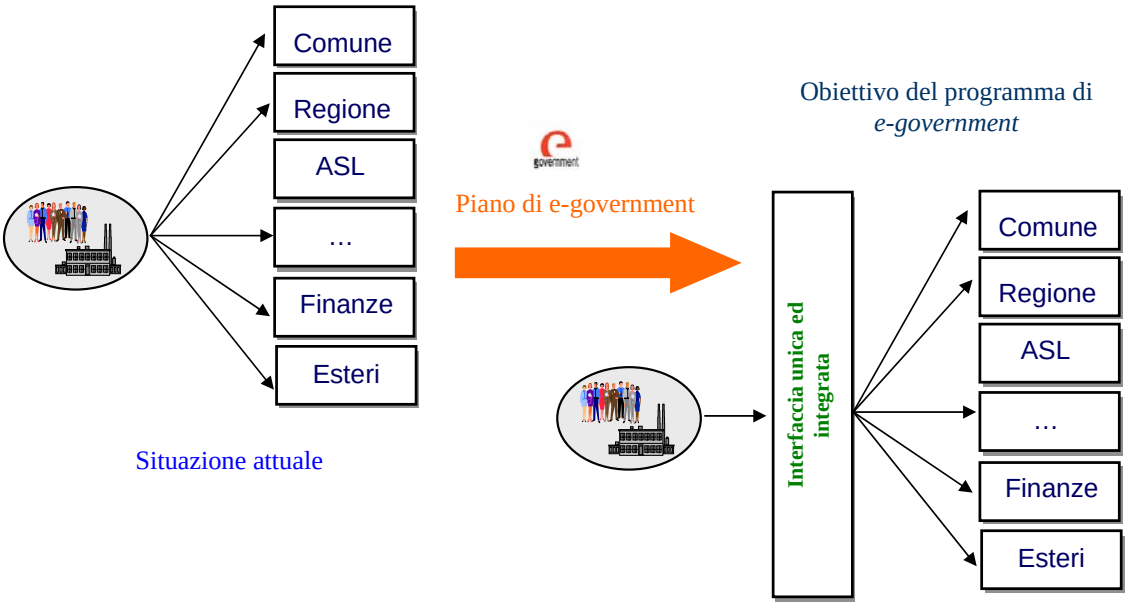
4. **Enti Eroganti**
   Un back office efficiente ed economicamente ottimizzato dei diversi enti eroganti

5. **Interoperabilità e cooperazione**
   Standard di interfaccia tra le Amministrazioni che consentano comunicazioni efficienti e trasparenza verso l'esterno.

6. **Infrastrutture di comunicazione**
   Una infrastruttura di comunicazione che colleghi tutte le Amministrazioni.

Il modello di e-government della Pubblica Amministrazione





14. E-Government: eventi e livelli di servizio

La fruizione dei servizi di e-government da parte di cittadini ed imprese sarà strutturata su una *metafora di comunicazione* modellata non sull'organizzazione della Pubblica Amministrazione, ma sulle effettive necessità dell'utenza.

La metafora si fonda sul concetto chiave di "**Evento della vita**": i servizi erogati dalle Amministrazioni locali saranno classificati in categorie specifiche per cittadini ed imprese.

Per i **cittadini** vengono proposte 15 categorie di "Eventi della vita":

1. Essere cittadino
2. Avere un figlio
3. Avere una famiglia
4. Vivere in salute
5. Abitare
6. Studiare
7. Lavorare
8. Percepire la pensione
9. Pagare le tasse
10. Fare e subire una denuncia
11. Usare un mezzo di trasporto
12. Vivere il tempo libero e la cultura
13. Fare sport
14. Andare all'estero
15. Vivere l'ambiente

Le Amministrazioni locali avranno la possibilità di estendere la metafora ad eventi caratteristici della realtà locale.

Per le **imprese** sono state definite 12 categorie di "Eventi della vita":

1. Aprire una nuova attività
2. Modificare un'attività
3. Sviluppare un'attività
4. Terminare un'attività
5. Finanziare un'attività
6. Gestire il personale
7. Possedere immobili
8. Pagare le tasse
9. Registrare marchi e brevetti
10. Importare ed esportare
11. Fare e subire una denuncia
12. Salvaguardare l'ambiente

Le Amministrazioni locali saranno tenute, per quanto possibile, a classificare i propri servizi nelle categorie proposte da questo modello, destinato a divenire uno standard di comunicazione per tutti i canali ed i mezzi di accesso alla Pubblica Amministrazione.

Un altro utile concetto nella comprensione del piano di e-government italiano è la definizione del "**livello di interazione**" di un servizio:

- **Livello 1**: informazioni di riferimento sulle procedure
- **Livello 2**: interazione *one-way* (es. download moduli)
- **Livello 3**: interazione *two-way* (es. avviare una procedura)
- **Livello 4**: esecuzione di una intera procedura (incluso l'eventuale pagamento online)

Questa definizione, mutuata dagli standard proposti dall'Unione Europea, indica alle Amministrazioni un metodo per valutare l'evoluzione dell'offerta telematica dei propri servizi.

- **Livello 1** ***informativo***
  Disponibili online solo le informazioni necessarie per avviare la procedura che eroga il servizio. Devono essere presenti le seguenti informazioni:
  - descrizione dell'organizzazione e delle attività dell'ente erogante
  - contatti per richiedere ulteriori informazioni (e-mail, telefono, etc.)
  - dettagli sulle procedure e modalità di erogazione del servizio.

- **Livello 2** ***download modulistica***
  È possibile trasferire online i moduli necessari ad avviare la procedura che eroga il servizio (interazione *one-way*). Si deve verificare *almeno una* delle seguenti condizioni:
  - possibilità di trasferire il modulo
  - possibilità di stampare il modulo
  - possibilità di richiedere il modulo (recapitato e rispedibile via posta ordinaria)

- **Livello 3** ***inoltro richiesta***
  È possibile avviare online la procedura che eroga il servizio (interazione *two-way*). Devono verificarsi *contemporaneamente* le seguenti condizioni:
  - esistenza di una procedura di autenticazione dell'utente
  - disponibilità del modulo elettronico che, compilato dall'utente, consente all'ente di avviare la procedura che eroga il servizio

- **Livello 4** ***esecuzione transazione***
  È possibile eseguire online l'intera procedura che eroga il servizio, comprensiva di eventuale pagamento, notifica e consegna. Devono verificarsi contemporaneamente le seguenti condizioni:
  - assenza di moduli cartacei
  - assenza della necessità di spostamenti fisici da parte dell'utente
  - possibilità di gestire online notifica, pagamento e consegna associati all'erogazione del servizio.

Il livello 4 non è previsto per i servizi che prevedono l'erogazione "fisica" di un servizio, come ad esempio la Carta di Identità o il Passaporto. In questi casi, come accade nelle transazioni di e-commerce, la fruizione del servizio online viene conclusa con la consegna fisica, accompagnata da una procedura informatica (ad esempio di tracciabilità).

Gli obiettivi del piano di e-government a breve termine (1-2 anni) sono:

- portare al livello 3 i servizi maggiormente utilizzati dagli utenti;
- sperimentare il livello 4 per i servizi prioritari.

Su un arco temporale più lungo (5-7 anni) l'obiettivo consisterà in un accesso completamente interattivo (livello 4) per la maggior parte dei servizi della Pubblica Amministrazione, sia locale che centrale.

La metafora di comunicazione del piano di e-government è una indicazione di come l'offerta di servizi della Pubblica Amministrazione dovrà evolversi nel breve/medio periodo ed è allo stesso tempo una sintesi dell'evoluzione di Internet nell'ultimo decennio.

La diffusione delle tecnologie ad un insieme sempre più vasto di aree applicative può essere schematizzato in una breve sintesi del percorso evolutivo della rete Internet:

- Inizialmente il diffondersi della TCP/IP protocol suite e di protocolli quali DNS, SMTP, FTP, NNTP, Gopher hanno permesso una visione unitaria della rete e lo scambio efficiente ed efficace di informazioni su scala planetaria.
- Successivamente l'avvento del protocollo HTTP e lo sviluppo dei primi browser hanno definito, con la nascita del World Wide Web, un modello, universalmente accettato, di distribuzione capillare dell'informazione.
- Infine l'attenzione si è spostata sulle applicazioni con il tentativo di applicare tecnologie quali DCOM, CORBA, RMI, Web Services allo sviluppo di applicazioni distribuite in grado di interoperare e cooperare.

Si può osservare come i livelli di interazione della metafora di comunicazione del piano di e-government non sfuggano a questa logica e in definitiva pongano le premesse per valutare la qualità dei servizi offerti dalla Pubblica Amministrazione in base a quanto siano vicini ad un paradigma applicativo distribuito.

15. E-Government: cooperazione applicativa

La struttura della Rete Nazionale e la metafora di comunicazione definita dal piano di e-government delineano uno scenario nel quale la *cooperazione applicativa* gioca un ruolo fondamentale ed in pratica vincola le amministrazioni ad implementare sistemi informatici in grado di interoperare.

Per meglio comprendere gli obiettivi ed i vincoli architetturali del piano di e-government è utile richiamare le definizioni e i concetti contenuti nella linee guida, relative alla cooperazione applicativa, del Centro Tecnico della Presidenza del Consiglio dei Ministri.

La Rete Nazionale è concepita come una ***federazione di domini***, dove per **dominio** si intende:

> "L'insieme delle risorse (procedure, dati, servizi) e delle politiche di una organizzazione"

ed è in primo luogo una definizione organizzativa che delinea il confine di responsabilità di un ente o di una amministrazione.

In questo modello la Rete Nazionale è il tramite attraverso il quale entità omogenee, i domini, sono in grado di comunicare e l'architettura cooperativa ha l'obiettivo di integrare gli oggetti informativi (procedure e dati) e le politiche di domini differenti.

Il punto chiave della interoperabilità applicativa diviene quindi definire in modo preciso le modalità che consentono ad un dominio *servente* di pubblicare i propri servizi affinché un dominio *cliente* possa accedervi. L'interoperabilità fra domini dovrà necessariamente fondarsi su standards definiti a livello nazionale e rispettare due vincoli fondamentali:

1. identificazione di dati e servizi che ogni amministrazione renderà disponibili;

2. definizione di politiche di sicurezza e di accesso e controllo della qualità e della correttezza dei servizi erogati.

Nel modello concettuale e organizzativo la cooperazione si realizza nella definizione di **porta di dominio**, l'elemento che consente l'accesso alle risorse applicative: la porta di dominio è un **adattatore** che consente ad un sistema informatico lo scambio di informazioni attraverso la Rete Nazionale, mediante messaggi che:

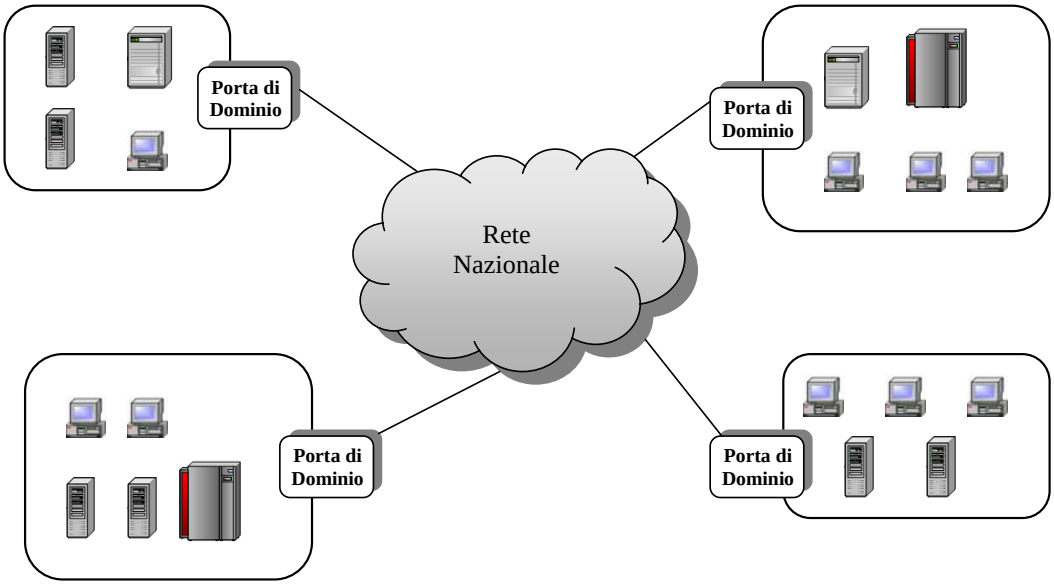
- sono **documenti XML** di formato concordato

- permettono la **comunicazione tra sistemi ed applicazioni** piuttosto che tra esseri umani

ed hanno due funzioni principali:

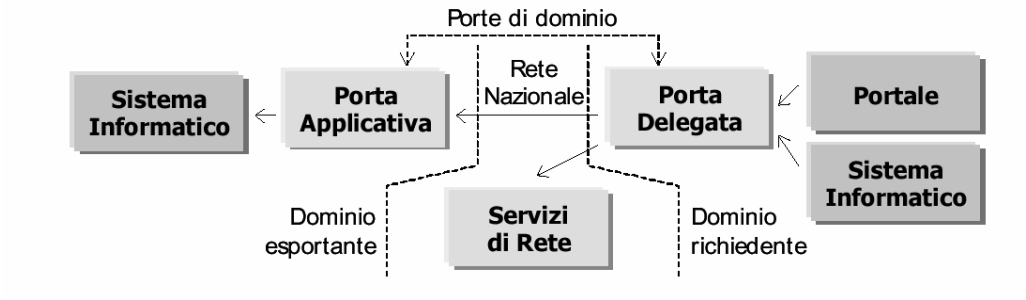
- **cooperazione** con altre porte di dominio

- **integrazione** verso l'interno del dominio.



Le porte di dominio possono assumere ruoli differenti e nella terminologia della Pubblica Amministrazione, vengono denominate:

- **Porta Applicativa**: una porta di dominio che interpreta il ruolo di *fornitore di un servizio*;
- **Porta Delegata**: una porta di dominio che interpreta il ruolo di *richiedente di un servizio*.



I domini si connettono alla Rete Nazionale attraverso le porte di dominio che isolano le responsabilità delle organizzazioni e rappresentano un'astrazione condivisa da tutte le tipologie di servizio.

In tale astrazione è possibile riconoscere un accenno del paradigma di interazione condiviso dalle definizioni di Web Services e Grid Services.

Oltre alle porte di dominio, per la cooperazione applicativa distribuita è necessaria la disponibilità di **servizi di rete** che forniscano funzionalità quali l'individuazione e la risoluzione degli indirizzi, il routing dei messaggi verso i domini destinatari e la comunicazione di eventi.

Il termine "sistema informatico" deve essere inteso in senso generale:

- un sistema centralizzato di una amministrazione di piccole dimensioni;
- un sistema distribuito sulla rete locale di una amministrazione medio/grande;
- una rete di area (aggregazione di comuni, comunità montana, rete provinciale o regionale) alla quale sono connesse i sistemi informatici di amministrazioni potenzialmente differenti (la rete territoriale gestita da un'agenzia o da un ministero).

Evidentemente la porte di dominio sono intese come una astrazione che permette di modellare l'interazione tra applicazioni. In tale modello le applicazioni possono essere pensate come collezioni, sempre nella terminologia della Pubblica Amministrazione, di *oggetti applicativi*.

Un **oggetto applicativo** di una applicazione è un insieme di strutture dati e di procedure predefinite che agiscono sui dati stessi. Nel paradigma *object oriented* potrebbe corrispondere alla definizione degli elementi e dei metodi di accesso di una classe.

L'interazione tra applicazioni può quindi essere ricondotta all'interazione tra oggetti applicativi e classificata in due tipologie fondamentali:

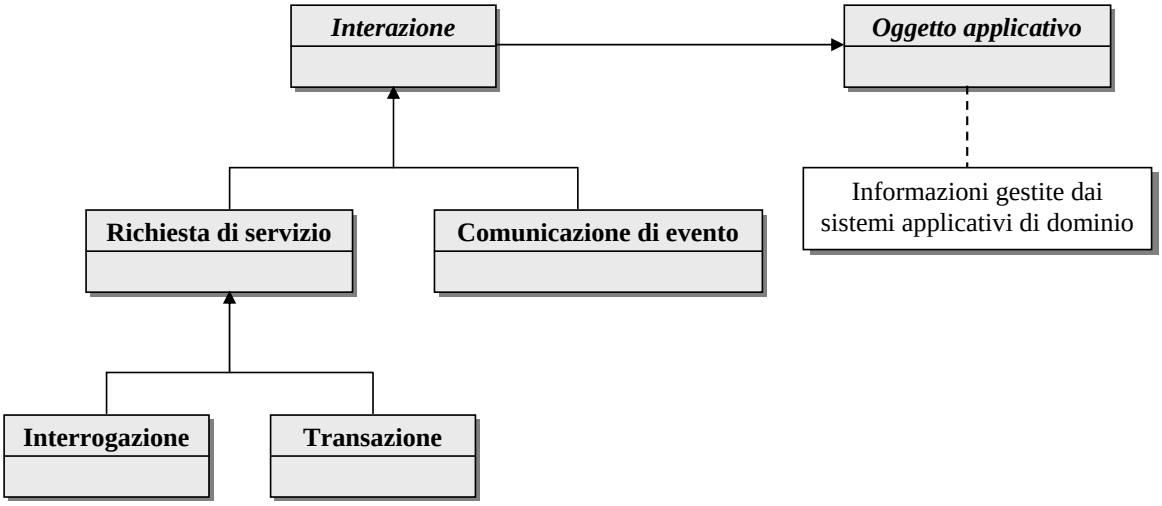
- **richiesta di servizio**: un messaggio prodotto da una applicazione di dominio cliente e diretto ad una applicazione di dominio servente. Il messaggio determina l'esecuzione di una procedura in un oggetto applicativo di una applicazione del dominio servente che, in base alle informazioni contenute nel messaggio, invia una risposta destinata all'applicazione cliente. La risposta è una indicazione *certa* che l'applicazione ha ricevuto ed eseguito la richiesta e fornisce le informazioni necessarie a determinarne l'esito, positivo o negativo.

  Dal punto di vista dell'interazione amministrativa la richiesta di servizio è sempre *sincrona*: l'applicazione cliente dovrà in ogni caso attendere l'esito della richiesta prima di procedere a successive nuove richieste.

  Le richieste di servizio possono essere classificate in due categorie:

  - ***interrogazioni***: acquisiscono informazioni da un dominio servente senza modificare in alcun modo lo stato di un oggetto applicativo di una applicazione del dominio;
  - ***transazioni***: generano una variazione permanente dello stato di almeno un oggetto applicativo di uno o più domini serventi. Il termini transazione è inteso in questo contesto nella sua valenza amministrativa, ma ha necessariamente anche implicazioni tecniche, quali, ad esempio, atomicità e reversibilità.

- **Comunicazione di evento**: un messaggio generato da una applicazione allo scopo di informare un insieme di applicazioni, di uno o più domini destinatari, di un cambiamento nello stato di un oggetto applicativo (es. cambio di residenza) o della creazione di un nuovo oggetto applicativo (es. registrazione di una nascita). Il messaggio dovrà contenere anche le informazioni che descrivono l'evento (modello push nella notifica di eventi dei Grid Services).



Dal punto di vista tecnico le interazioni applicative prevedono lo scambio di una coppia di messaggi: una richiesta inviata ad una porta delegata a cui fa seguito una risposta da parte di una porta applicativa. In questo senso l'interazione applicativa raccomandata nelle definizioni della cooperazione applicativa è sempre *sincrona*. D'altra parte le stesse definizioni introducono una distinzione tra scambio di messaggi ***sincrono*** ed ***asincrono***, dal punto di vista di una porta applicativa:

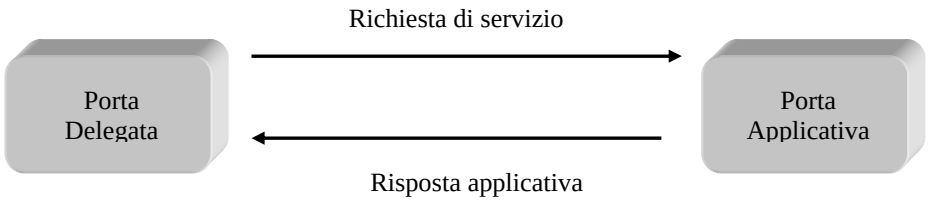
- in uno scambio **sincrono** la porta delegata invia la propria richiesta applicativa ed attende la risposta dalla porta applicativa;

- in uno scambio **asincrono** la risposta della porta applicativa può essere inviata in un tempo successivo e la porta delegata non rimane in attesa.

I termini sincrono ed asincrono, in queste definizioni, non sono intesi quindi dal punto di vista dei protocolli di comunicazione utilizzati, ma dal punto di vista degli oggetti applicativi. È un punto importante che merita di essere ulteriormente approfondito.

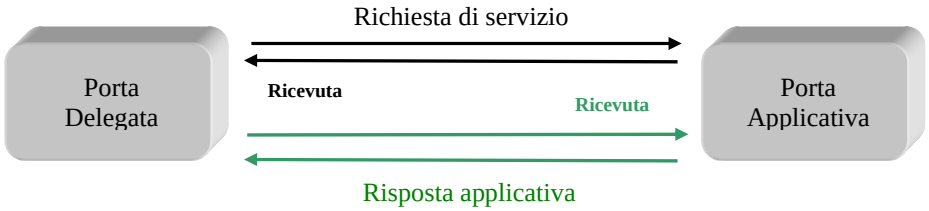
In una interazione **sincrona** la porta delegata di un dominio cliente invia una richiesta di servizio ad un oggetto applicativo, raggiunto mediante una porta applicativa di un dominio servente, il quale restituisce una risposta applicativa contenente i dati richiesti e con ciò si esaurisce l'interazione con il servizio.

Interazione sincrona

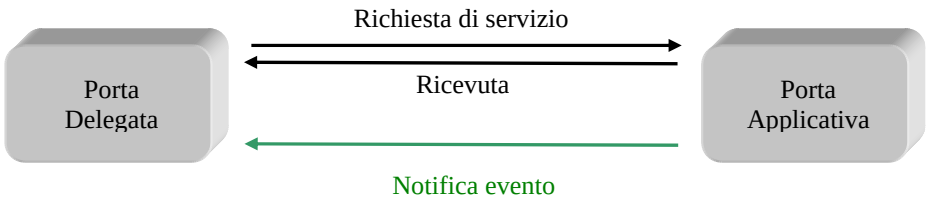


In una interazione **asincrona** la risposta alla richiesta di servizio è in generale costituita da una **ricevuta** che conferma la ricezione della richiesta e non conclude necessariamente l'interazione. L'interazione può quindi essere portata a compimento successivamente o in modo sincrono, fornendo i dati contenuti nella ricevuta, o con la comunicazione di un evento.

Interazione asincrona



Interazione asincrona



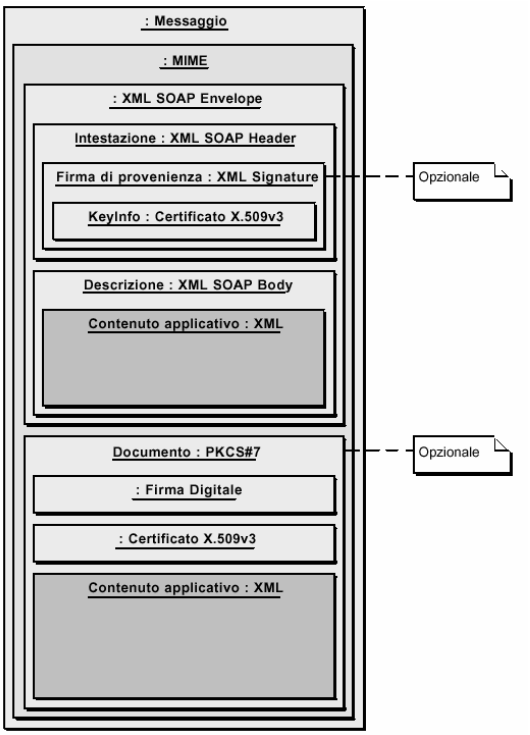
Nei principi di cooperazione gli schemi di interscambio di messaggi sono denominati "**Profili di collaborazione**" e si sottolinea che la scelta tra un profilo sincrono o asincrono può dipendere da aspetti legati alla latenza delle procedure amministrative: la necessità di un intervento umano, l'apposizione di una firma digitale di un pubblico ufficiale, etc. Tuttavia anche considerazioni e razionali puramente tecnologici possono condurre a decidere in una direzione piuttosto che nell'altra.

I protocolli di trasporto raccomandati nei principi di cooperazione applicativa sono "*protocolli leggeri basati su codifiche testuali*" ed in particolare viene consigliato di utilizzare il protocollo HTTP per la comunicazione sincrona ed asincrona interdominio e/o il protocollo SMTP per lo scambio asincrono di messaggi. Viene inoltre raccomandato di ridurre al minimo indispensabile l'uso di protocolli crittografici (come ad esempio HTTP/S), da utilizzare esclusivamente per la trasmissione di dati sensibili.

L'universale disponibilità dei protocolli di trasporto HTTP, HTTP/S ed SMTP ed il fatto che siano implicitamente in grado di attraversare i sistemi di sicurezza, inevitabilmente posti a protezione di un dominio, è, ancora una volta, uno dei razionali che sembra aver guidato la scelta delle "Modalità di scambio telematico" definite negli allegati tecnici del piano di e-government.

L'allegato tecnico n.2 "*Rete Nazionale: caratteristiche e principi di cooperazione applicative*", dal quale queste note sono state tratte, fornisce chiare ed inequivocabili indicazioni sull'uso di linguaggi fondati su grammatiche XML per modellare la definizione e lo scambio di dati. In particolare il protocollo SOAP viene descritto come "*lo standard emergente per il veicolamento delle informazioni codificate con XML sulla rete Internet, mediante il protocollo HTTP*".

L'XML, il paradigma distribuito Web Services, i protocolli SOAP ed HTTP sembrano quindi essere la struttura fondante dei principi di cooperazione applicativa del piano di e-government che si spinge fino al punto di fornire indicazioni sulla codifica dei messaggi SOAP da utilizzare, per implementare sotto forma di Web Services, i servizi esposti dalle porte applicative e delegate delle amministrazioni. Tali indicazioni prendono forma nella definizione della "***busta di e-government***"



L’allegato tecnico prescrive genericamente intestazioni del messaggio HTTP o SMTP, ma lascia aperta la possibilità, come previsto dal protocollo SOAP, all’uso di altri protocolli di trasporto.

Per il resto la busta di e-government è un messaggio SOAP, pienamente compatibile con gli standard prescritti nei Web Services o nell’architettura OGSA, costituito da due parti:

- **intestazione** (SOAP Header) contenente dati relativi al messaggio stesso, in particolare l’identificativo del messaggio, e la firma digitale (opzionale) secondo gli standard *XML SOAP Security* e *XML Segnature*;

- **descrizione** (SOAP Body) ovvero il contenuto applicativo del messaggio.

Il messaggio SOAP può a sua volta essere incluso in un messaggio MIME allo scopo di allegare al messaggio uno o più documenti applicativi, secondo lo standard *XML SOAP with attachments*: ad esempio potrebbe essere allegato un documento su cui è stata apposta la firma di un pubblico ufficiale.