Web Services in pratica:

Esperienza nello sviluppo con framework gSOAP

Incontro CCR—LNL, 10 dicembre 2008

Alvise Dorigo
INFN Padova
alvise.dorigo@pd.infn.it

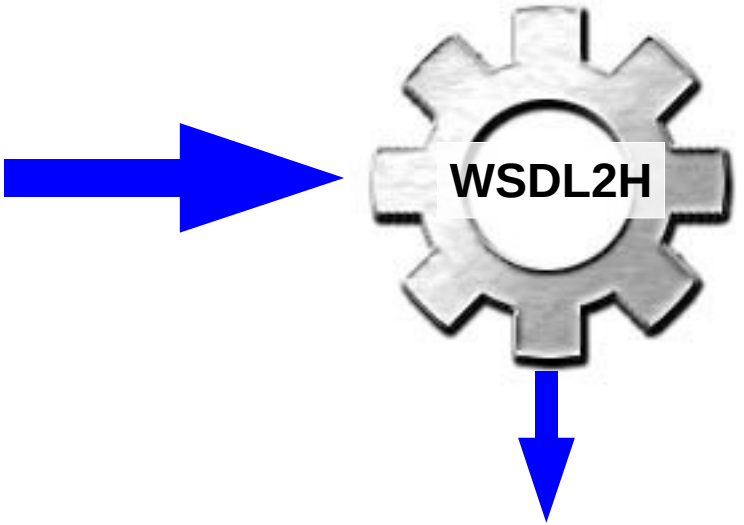
Ruolo di un framework WebService di sviluppo

- Un framework per WebService (WS) è una **libreria di API** e **tools di sviluppo** che processa la descrizione di un WS scritta in linguaggio WSDL (sintassi XML)
- Il framework produce uno **stub** fatto di:
  - Classi che rappresentano i dati di input e di output delle funzioni del servizio remoto
  - 'Signatures' di funzioni o metodi di classe che il client deve invocare per dialogare col WS
  - La comunicazione SOAP di basso livello è nascosta; l'utente ignora le problematiche di [de-]serializzazione XML e comunicazione di rete sul trasporto HTTP[s].
- Lo sviluppatore può quindi concentrarsi **solo sulla logica di alto livello** ("*logica businnes*") semplicemente inserendo codice negli stub.

In pratica...

```
<xsd:element name="JobRegisterRequest">
  <xsd:complexType>
  <xsd:sequence>
  <xsd:element name="jobDescriptionList"
         type="cream_types:JobDescription"
         MinOccurs="1"
        maxOccurs="unbounded" />
  </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Esempio di dato di input
per la funzione JobRegister
di un CE WS.

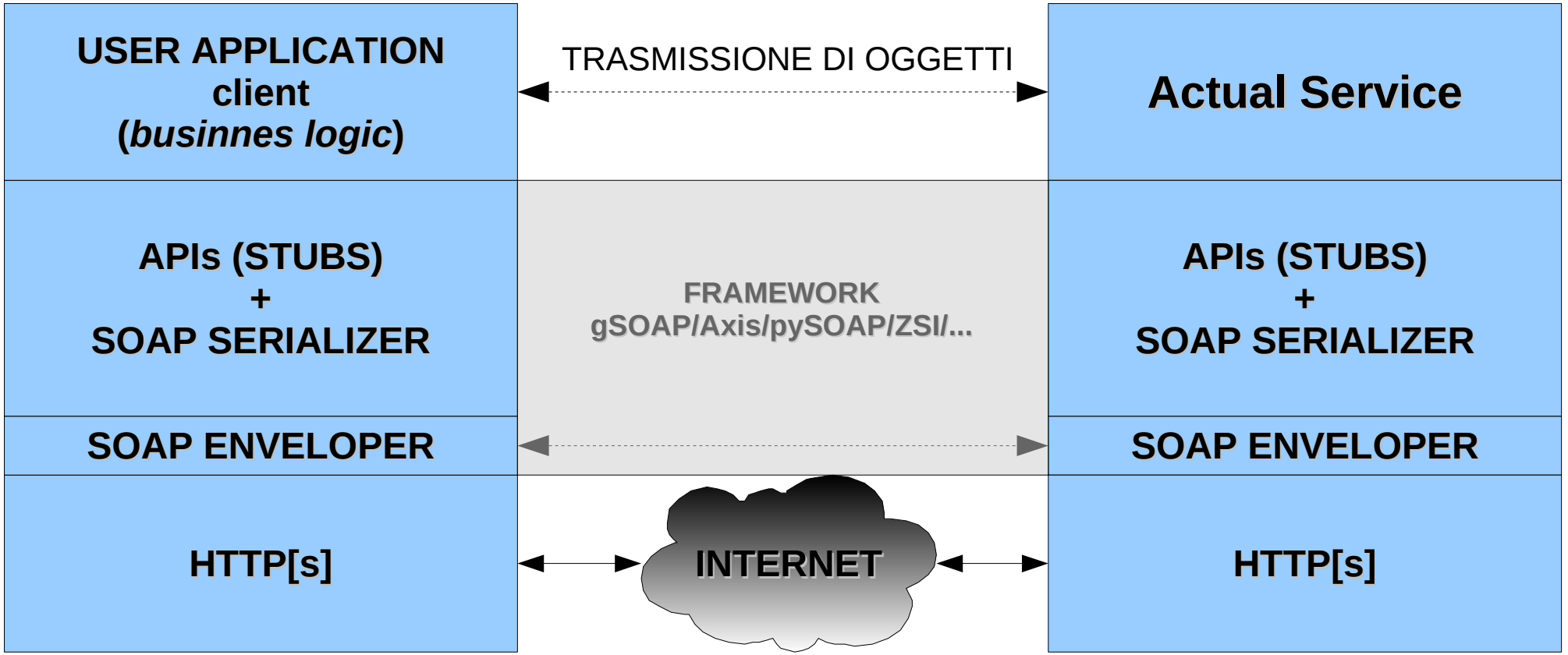


Tool command line
del framework
gSOAP

```
class SOAP_CMAC _CREAMTYPES__JobRegisterRequest
{
public:
        std::vector<CREAMTYPES__JobDescription * >jobDescriptionList;
        struct soap *soap;      /* transient */
public:
        virtual int soap_type() const { return 27; }
        virtual void soap_default(struct soap*);
        virtual void soap_serialize(struct soap*) const;
        virtual int soap_put(struct soap*, const char*, const char*) const;
        virtual int soap_out(struct soap*, const char*, int, const char*) const;
        virtual void *soap_get(struct soap*, const char*, const char*);
        virtual void *soap_in(struct soap*, const char*, const char*);
                _CREAMTYPES__JobRegisterRequest() { }
        virtual ~_CREAMTYPES__JobRegisterRequest() { }
};
```

Sua renderizzazione
in C/C++ da parte di un
framework

Stack


USER APPLICATION
client
(*businnes logic*)
TRASMISSIONE DI OGGETTI
Actual Service
APIs (STUBS)
+
SOAP SERIALIZER
FRAMEWORK
gSOAP/Axis/pySOAP/ZSI/...
APIs (STUBS)
+
SOAP SERIALIZER
SOAP ENVELOPER
SOAP ENVELOPER
HTTP[s]
INTERNET
HTTP[s]

Know-How

- Definire a livello di WSDL le funzionalità del servizio esposte su WEB (e i dati manipolati):
  - acquisizione di know-how sull'XML e sulla specifica per la descrizione dei WS (WS-I compliance, regole per l'interoperabilità tra framework/linguaggi/piattaforme diversi)
- Utilizzare i tool del framework per produrre gli stub nel linguaggio target (Java, C++, python, etc.)
  - Scelta di un framework che possa produrre stub in un linguaggio target ben noto (**gSOAP/Axis-cc** per il C/C++, **Axis-Java** per Java, **SOAPpy/ZSI** per python)
- Studiare lo stub prodotto e procedere all'implementazione del corpo delle funzioni
- Per un primo prototipo dummy, 1 **mese uomo**, senza alcuna conoscenza di WS ma assumendo almeno una **buona conoscenza del linguaggio target**

Esperienza con Axis-Java/gSOAP 1/3

- Framework OpenSource se si è “in budget”: **Axis** genera stub in Java e C/C++, **gSOAP** solo C/C++.
- gSOAP produce codice più “difficile” da leggere (rispetto ad Axis):
  - Poco object oriented. Scarso uso di container templetizzati e ampio uso di puntatori => memory leaks/corruptions. Classi con tutti i membri pubblici.
  - Una vecchia versione non dichiarava *virtual* il distruttore di classi base => **impossibilità di utilizzare ereditarietà** => bisognava scriversi dei wrapper!
  - Nessun metodo get/set => derivare per forza le classi autogenerate allo scopo di nascondere i dati (il cui nome/namespace può cambiare con la versione del fwk)
  - In certi casi, poco frequenti, per problemi tecnici si è reso necessario wrapp-are le classi base autogenerate (molto codice da scrivere)
- gSOAP complica la gestione dei faults ritornati dal servizio. Necessità di usare codice condizionale (`if/switch-case`) invece che un comodo approccio con le eccezioni (supportate dai principali linguaggi OO).
- Cripticità di gSOAP negli errori di basso livello (connessione di rete col servizio):
  - non sempre si è in grado di capire la reale natura di una connessione fallita (nome host sbagliato, host spento, nome del context del servizio errato, etc.). **Bug ? Risolto in recenti versioni ?**

Esperienza con Axis-Java/gSOAP 2/3

- gSOAP ha una documentazione completa ma povera di esempi e casi reali, soprattutto per quanto riguarda le situazioni d'errore.
  - Ma comunque sufficiente l' “impaziente” che vuole realizzare velocemente un servizio e un client molto semplici.
- Problemi di interoperabilità gSOAP/Axis-Java hanno imposto numerose revisioni del WSDL e passaggi dal formato *doc-literal* al *rpc-literal* e viceversa prima di capire la causa.
- Supporto pressoché nullo da parte degli sviluppatori di gSOAP (a pagamento ?)
- Cambio di versione di gSOAP spesso **non backward-compatible!!** Abbiamo rinunciato ad una recente migrazione a gSOAP 2.7.10 (per via di cambiamenti pesanti nel codice autogenerato).
- gSOAP genera **codice VELOCISSIMO.**
- Axis-cc genera codice più “friendly” e più object oriented (e.g. exceptions). Meno tempo da passare sull'analisi dello stub autogenerato. Meno codice da scrivere (classi più “autonome”).

Esperienza con Axis-Java/gSOAP 3/3

- WS molto comodi: multi-platform, facilità di debug a runtime:
  - basta un echo server per vedere i dati ASCII scambiati fra client e servizio.
- Abbiamo fatto controlli incrociati per lo studio dell'interoperabilità (difficile se il protocollo fosse stato binario):
  - Dump della comunicazione client C++ → servizio C++ (gSOAP vs gSOAP)
  - Dump della comunicazione client C++ → servizio Java (gSOAP vs Axis)
  - Confronto dei dump per studiare i problemi di interoperabilità

Security

- Possibilità di compilare con abilitazione SSL
- Possibilità di inserire un proprio plug-in per comunicazioni crittografate (o per qualunque altra extra-manipolazione dei dati input/output)
- La comunicazione non è più in chiaro. Non basta più un banale echo server.
  - Dopo lo switch definitivo a comunicazione sicura tempo fa sviluppammo un ssl-echo server per vedere cosa si scambiavano client e server (i problemi di interoperabilità sono sempre in agguato soprattutto se si cambiano versioni dei framework o se si fanno minime modifiche al WSDL del servizio)

Alternative ?

- Java: linguaggio e VM mature e veloci. Axis è più robusto e amichevole. Più ripida la curva di apprendimento.
- Java (come ogni linguaggio [semi-]interpretato) gira ovunque e la perdita di prestazioni rispetto al nativo C/C++ è irrilevante in un ambiente WS.
  - La comunicazione in ambienti WS è intrinsecamente lenta e onerosa ([un-]parsing di XML e overhead SOAP), soprattutto se si usa il substrato SSL.
- Sembra ci siano meno problemi di interoperabilità usando Axis verso altri framework (non gSOAP)
- gSOAP ostico anche in fase di compilazione (molti flag di compilazione per abilitare/disabilitare funzionalità importanti)

Perchè i WebService ?

- Nella nostra esperienza disaccoppiamento totale fra logica e comunicazione (API consolidate)
- Multi-platform: il server CREAM **in linea di principio** può girare su Windows, i suoi client su Unix (e viceversa).
- “Moderno”: tecnologia nuova e molto diffusa, basata su **principio SOA (service oriented architecture)**, molto utilizzato per integrare applicazioni legacy in sistemi moderni.
  - Abbiamo un CE che espone su WEB i suoi servizi e pubblica il WSDL → chiunque può realizzarsi un client per sottomettere job da qualsivoglia piattaforma.
- Protocollo testuale su HTTP: di default aperto su tutti i firewall di macchine che offrono servizi web.