RPC & Web Services

Benjamin Canou - Christian Queinnec
Master 2 Informatique UPMC - Spécialité STL

Cours 2 du 26/11/2012

Architecture des Applications Réticulaires

À l'origine : RPC

Fonctionnement général
Historique & Tour d'horizon

À l'origine une norme (RFC 1057), aujourd'hui un terme générique.
Principe général :

- Un serveur fournit un service
- Appelable depuis un client
- Prenant éventuellement des paramètres
- Retournant éventuellement une valeur (fonction distante)

Sont en général associés :

- Un annuaire des services / fournisseurs de services
- Une interface typée de chaque service
  - Non liée au langage (souvent : procédural + types simples)
  - Liée au langage (souvent : objet + types objet)
- Une intégration au langage : appel distant $\equiv$ appel normal

À quoi ça sert ?

- Programmation répartie
- Communication inter-processus
- Et bien sur : Services Web

Mécanisme très semblable à un appel de fonction local :

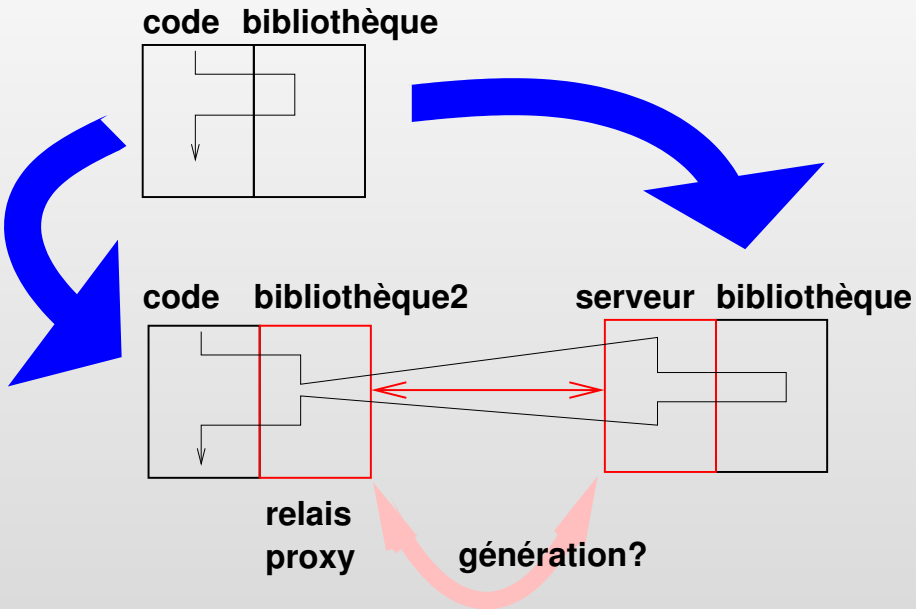
- Valeurs $\rightarrow$ endroits connus
- Saut vers appelé
- Résultats $\rightarrow$ endroits connus
- Reprise appelant

Mais il y a de nombreuses difficultés techniques :

- Non immédiateté des transferts d'information
- Risque d'erreur dans la propagation
- Copie des données
- Coût des transmissions
- Niveau de sécurité différent

Donc on repose sur des briques de base saines :

- Représentation des données bien définie et portable
- Langage de description d'interfaces (IDL) typé
- Génération automatique de code



On veut rendre accessible à distance une bibliothèque.

- Sans changer son implantation
- Sans changer le code du programme

Donc on insère du code des deux côtés :

- Côté client, une bibliothèque qui présente la même interface et transmets les appels
- Côté serveur, un recepteur d'appels qui appelle la bibliothèque originale

Code original :
Bibliothèque originale :

```
1 : z = f(x, y);
2 : function f (x, y) { return /* ... */ ; }
```

Code serveur à écrire :

```
1 : function process_invocation () {
2 :   var fname = receive (), args = receive ()
3 :   if ( fame == "f" ) {
4 :     send (encode(f(decode(args[0]), decode(args[1]))));
5 :   } else if /* ... */
6 : }
7 : while (true) process_invocation ();
```

Nouvelle bibliothèque client à écrire :

```
1 : function f (x, y) {
2 :   send ("f");
3 :   send ([encode (x), encode (y)]);
4 :   return decode(receive ());
5 : }
```

La magie est dans : encode, decode, send et receive, et on ne veut pas écrire ce code

Technologies RPC

Systèmes complets et briques indépendantes
Critères de choix

Architecture des Applications Réticulaires

1982: Sun introduit NFS, "The network is the computer", avec RPC et XDR.

À l'ancienne : XDR (eXternal Data Representation), sous couche de RPC

- Tout est transmis par mots de 4 octets
- L'ordre des octets est fixé
- Types de données C
  (entiers, flottants, caractères simples, répétitions, structures, tableaux)

Sérialisation des langages modernes :

- Valeur du langage $\leftrightarrow$ suite d'octets
- Gestion du partage, des cycles
- Chargement dynamique de code / classes / modules

Pour échanger des données, on sérialise, mais :

- Il faut assurer la sécurité d'exécution :
  - Que le client n'appelle que des fonctions du serveur autorisées
  - Que le serveur vérifie les types des paramètres
  - Que le client vérifie les types des résultats

  On peut alors :
  - Générer les vérifications à partir d'un IDL
  - Reposer sur une sérialisation sûre
  - Utiliser la réflexion
- Il faut savoir traiter les pointeurs de code :
  - Reposer sur un mécanisme d'annuaire
    pointeur de code = appel distant (callbacks)
  - Transmettre du code à charger dynamiquement

Exemple d' IDL bas niveau : ASN.1 (Abstract Syntax Notation One)

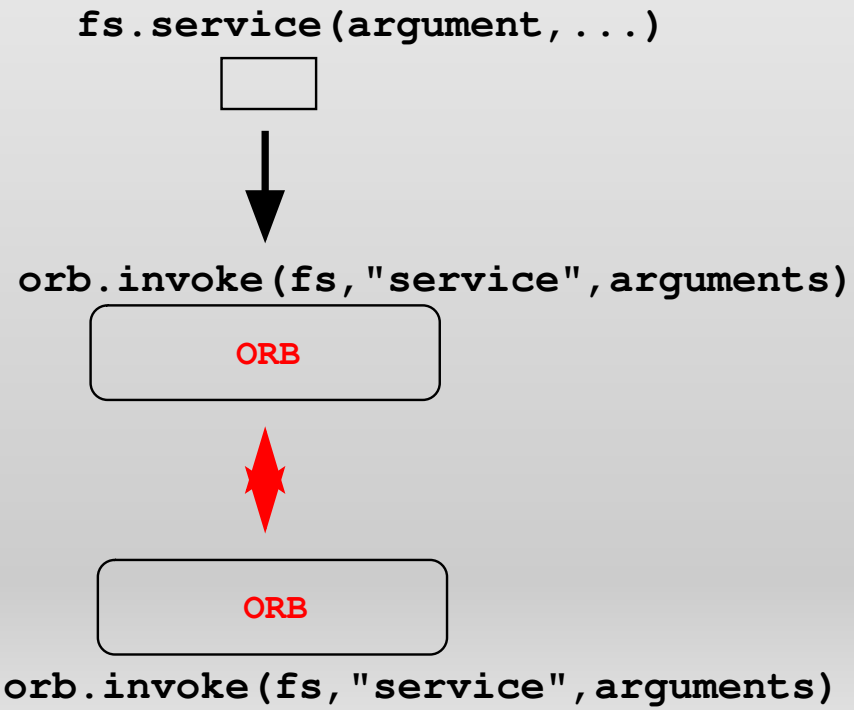
- Un langage de description de données (et de signatures de fonctions).
- Types de base: entier, flottant, booléen, caractère.
- Composeurs de types: enregistrement, répétition, union.
- Ne fixe pas le format de sérialisation (ex. BER, DER).
- Utilisé dans les normes de sécurité et télécoms.
- Utilisé pour les certificats X509 (navigation HTTPS).

Exemple d' IDL haut niveau : Mozilla IDL

- Description d'objets (méthodes, champs)
- Hiérarchie de classes
- Paramètres, valeurs de retours et champs typés
- Paramètres d'entrée et de sortie
- Utilisé entre le moteur (C++) et l'interface (JavaScript)

Exemples d'intergiciels : Corba, DBUS, DCOM

- Fournisseur de services distants et partagés (IPC).
  ORB (Object Request Broker)
- Indépendance vis-à-vis des langages de programmation (IDL).
- Fonctions d'introspection et découverte.



Java et RMI

- RMI ≈ RPC en et pour Java
- IDL ≈ Interface (compatibilité Corba avec idl2java, java2idl)
- Service associé d'annuaire JNDI

Sérialisation ayant de nombreuses propriétés supplémentaires

- Particularisable (par surcharge)
- Transmission de classes (nom et/ou code)
- Gestion des versions des classes
- Resynchronisable

XML et SOAP

- XML: un langage parenthésé décrivant des arbres régis par une grammaire (DTD, XML schéma, RelaxNG). Pourquoi donc ne pas échanger en XML ?
- Transmission de l'appel en XML :

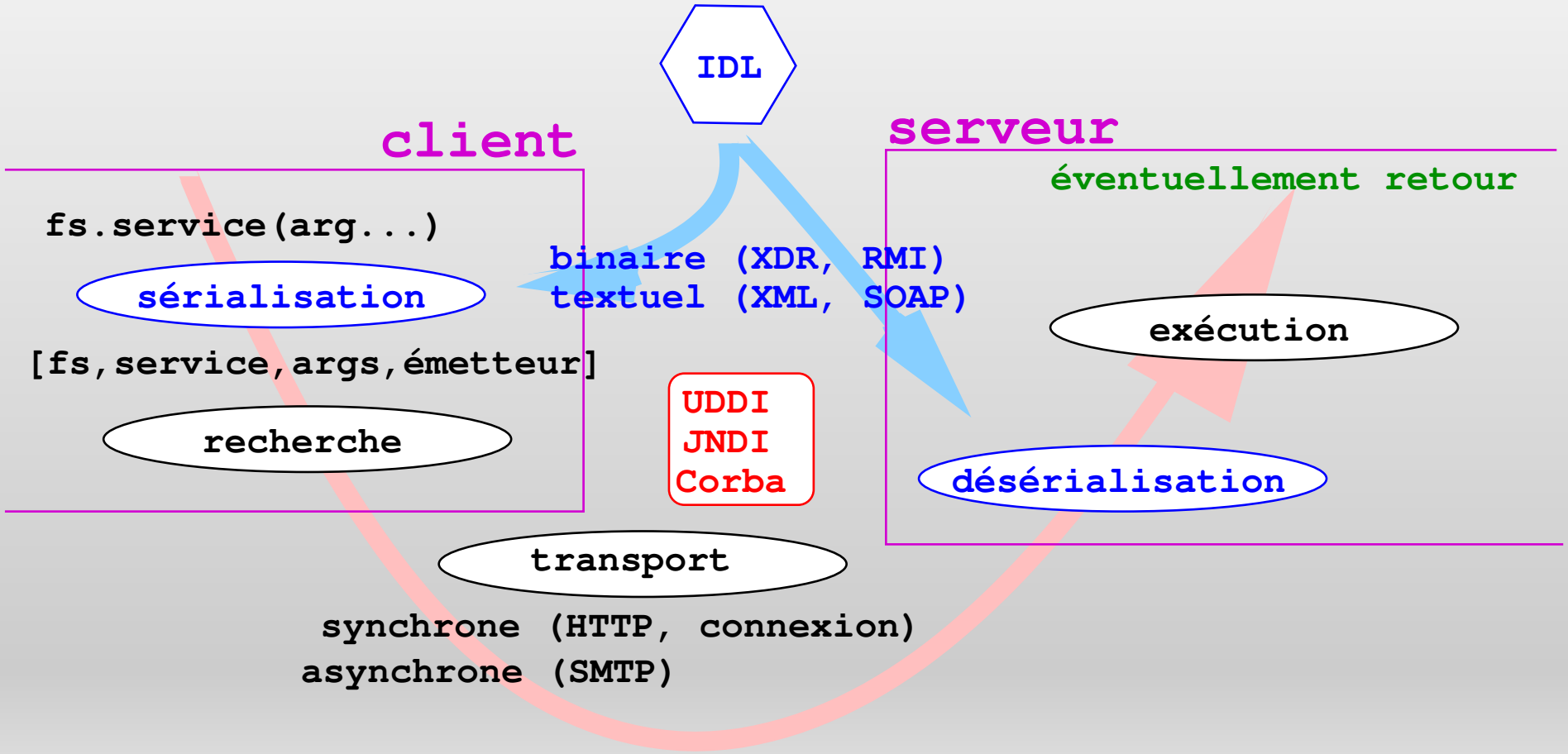
```
1 : <f>
2 :    <x>3.14</x>
3 :    <y>2</y>
4 : </f>
```

- Transmission du résultat en XML :

```
1 : <fresult>—1.0</fresult>
```

1998: XMLRPC → 1999: SOAP → 2005: JBI (XML + pièces jointes)

Schéma récapitulatif



Choix du formalisme

Comment décrire les services ?

- Interface en Java
  (compilateurs idl2java, java2idl)
- Description en IDL
  (compilateurs de codeur/décodeur vers langages de programmation à la Corba)
- Descripteur XML (WSDL)
  (compilation/interprétation des descripteurs)
- Annotations dans code Java, C# pour engendrer le précédent descripteur
- Utilisation des capacités réflexives pour ne même plus annoter

Soucis sur certains types de données.

Choix de la sérialisation

- Comment coder/empaqueter l'information ?
  - Binaire : efficace
    mais difficilement déployable, couplage client/serveur, peu évolutif
  - Textuel : verbeux
    mais universel, déployable (print/read), assez évolutif, relayable, vérifiable
- Soucis pour pointeurs, fonctions, … sans oublier l'émetteur!
- Autres considérations:
  - Sécurité, signature, chiffrement
  - Intégrité (résistance aux pannes), resynchronisation,
  - Gestion des versions client/serveur, auto-description
  - Qualité de service, priorité, relais,

Choix de la recherche

Où se trouve un fournisseur de services ?

- Ports bien connus
- Sites bien connus (DNS)
- Annuaire caché dans les ORB (Corba)
- Annuaire UDDI, JNDI, JINI, LDAP, UPnP, JBI

Dans tous les cas, un fournisseur de services s'enregistre sous un nom dans un annuaire par lequel il pourra être retrouvé, il spécifie aussi comment on doit lui parler.

Choix de couplages

- Du côté du langage du serveur, on peut souhaiter voir:
  - Un objet Request lié au cadre employé (Axis par ex.)
  - Un DOM (ou un SAX dégraissé)
  - Un POJO (*Plain Old Java Object*)
- Idem côté client.

Choix du transport

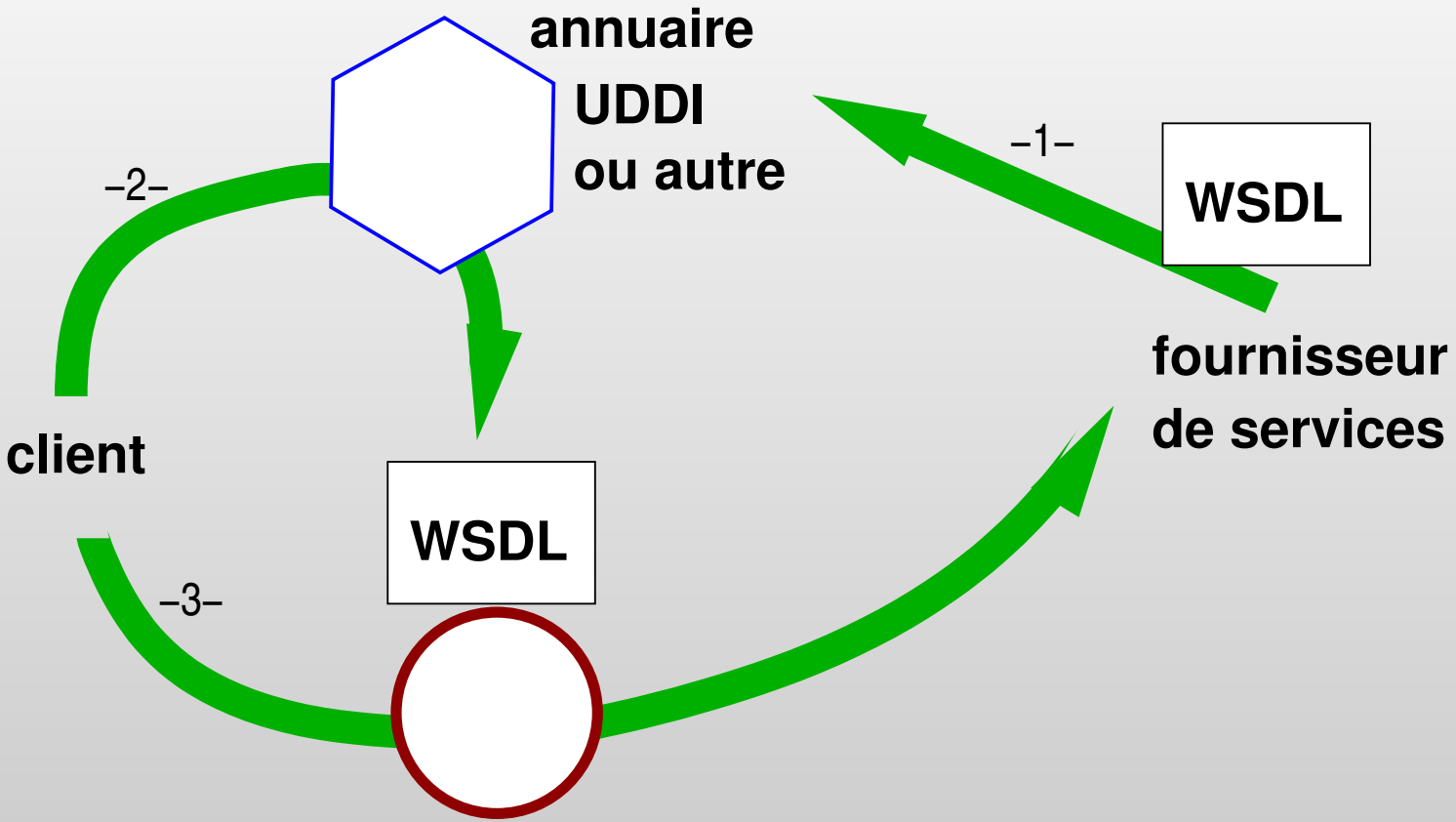
- Synchrone (HTTP, FTP, connexion) ou asynchrone (SMTP, RSS)
- Avec retour ou pas
- Répétition : une fois, au plus une fois, au moins une fois

Application aux Web Services

Description d'un service SOAP en WDSL
Exemple de requête en SOAP
Appels distants en Perl & Java

Architecture des Applications Réticulaires

Usage d'un descripteur WSDL



L'initiative UDDI a été abandonnée. La génération de la bibliothèque d'exécution peut être interprétée (Perl) ou compilée (wdsl2java).

Concepts WSDL

- service: une collection de points d'accès (*port*)
- port: une adresse réseau (numéro IP, URL) et un couplage
- couplage ou binding: un protocole (HTTP, SMTP, etc.) et un format de données (SOAP, une grammaire XML, etc.) pour chaque opération (et chacun de ses messages)
- Les opérations définissent des échanges de messages (requête seule, requête/réponse) et sont regroupées dans les portTypes. Dans JBI, on a les conversations complètes.
- chaque message définit sa signature c'est-à-dire ses données et leur type.
- Enfin le tout est imbriqué dans des définitions

Début du fichier (tiré de la doc. W3C) :

- Espaces de noms classiques
- targetNamespace : URI de ce document WSDL

```
1 : <?xml version="1.0"?>
2 : <definitions
3 :  name="StockQuote"
4 :  targetNamespace="http://fictif/stockquote.wsdl"
5 :  xmlns:tns="http://fictif/stockquote.wsdl"
6 :  xmlns:xsd1="http://fictif/stockquote.xsd"
7 :  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
8 :  xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Types de données, ici décrits en XMLSchema :

```
1 :   <types>
2 :     <schema targetNamespace="http://fictif/stockquote.xsd"
3 :             xmlns="http://www.w3.org/2000/10/XMLSchema">
4 :       <element name="TradePriceRequest">
5 :         <complexType>
6 :           <all><element name="tickerSymbol"
7 :                         type="string"/></all>
8 :         </complexType>
9 :       </element>
10 :      <element name="TradePrice">
11 :        <complexType>
12 :          <all><element name="price"
13 :                        type="float"/></all>
14 :        </complexType>
15 :      </element>
16 :    </schema>
17 :  </types>
```

Messages correspondant à l'appel et au retour :

```
1 :   <message name="GetLastTradePriceInput">
2 :     <part name="body" element="xsd1:TradePriceRequest"/>
3 :   </message>
4 :   <message name="GetLastTradePriceOutput">
5 :     <part name="body" element="xsd1:TradePrice"/>
6 :   </message>
```

Combinaison de l'appel et du retour dans une opération :

```
1 :   <portType name="StockQuotePortType">
2 :     <operation name="GetLastTradePrice">
3 :       <input message="tns:GetLastTradePriceInput"/>
4 :       <output message="tns:GetLastTradePriceOutput"/>
5 :     </operation>
6 :   </portType>
```

Liaison transport SOAP sur HTTP :

- Adresse (URL puisque HTTP)
- Encodage des contenus, ici le XML verbatim (literal)

```
1 :   <binding name="StockQuoteSoapBinding"
2 :            type="tns:StockQuotePortType">
3 :     <soap:binding style="document"
4 :      transport="http://schemas.xmlsoap.org/soap/http"/>
5 :     <operation name="GetLastTradePrice">
6 :       <soap:operation
7 :         soapAction="http://fictif/GetLastTradePrice"/>
8 :       <input>
9 :         <soap:body use="literal"/>
10 :      </input>
11 :      <output>
12 :        <soap:body use="literal"/>
13 :      </output>
14 :    </operation>
15 :  </binding>
```

Finalement, la définition du service :

```
1 :    <service name="StockQuoteService">
2 :      <documentation>My first service</documentation>
3 :      <port name="StockQuotePort"
4 :            binding="tns:StockQuoteSoapBinding">
5 :        <soap:address
6 :         location="http://fictif/stockquote"/>
7 :      </port>
8 :    </service>
9 : </definitions>
```

On a vu un exemple très simple, il est bien sûr possible :

- De multiplier les messages, opérations, etc.
- De les combiner de sifférentes façons

Exemple requête SOAP

```
1 : <SOAP-ENV:Envelope
2 :    xmlns:SOAP-ENV= "http://schemas.xmlsoap.org/soap/envelope/"
3 :    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/enco
4 :    <SOAP-ENV:Body>
5 :       <m:getLastTradePrice
6 :            xmlns:m="http://fictif/stockquote.xsd">
7 :         <tickerSymbol>IBM</tickerSymbol>
8 :       </m:getLastTradePrice>
9 :    </SOAP-ENV:Body>
10 : </SOAP-ENV:Envelope>
```

Exemple réponse correcte SOAP

```
1 : <SOAP-ENV:Envelope
2 :    xmlns:SOAP-ENV= "http://schemas.xmlsoap.org/soap/envelope/"
3 :    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/enco
4 :    <SOAP-ENV:Body>
5 :       <m:getLastTradePriceResponse
6 :            xmlns:m="http://fictif/stockquote.xsd">
7 :         <price>42</price>
8 :       </m:getLastTradePriceResponse>
9 :    </SOAP-ENV:Body>
10 : </SOAP-ENV:Envelope>
```

Exemple réponse erronée SOAP

```
1 : <SOAP-ENV:Envelope
2 :    xmlns:SOAP-ENV= "http://schemas.xmlsoap.org/soap/envelope/"
3 :    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/enco
4 :    <SOAP-ENV:Body>
5 :       <SOAP-ENV:Fault>
6 :          <faultCode>SOAP-ENV:Client</faultCode>
7 :          <faultString>...</faultString>
8 :       </SOAP-ENV:Fault>
9 :    </SOAP-ENV:Body>
10 : </SOAP-ENV:Envelope>
```

On ne s'intéresse qu'au côté client.

- Lignée XMLRPC:
  - Nécessité d'un analyseur XML (expat natif ou XML::Parser en Perl)
  - RPC::XML $\neq$ XML::RPC (merci les gars)
- Lignée SOAP:
  - SOAP::Lite

Exemple d'appel du service SOAP précédent :

```
1 : use SOAP::Lite;
2 : print SOAP::Lite
3 :  ->service('http://fictif/StockQuote.wsdl')
4 :  ->getLastTradePrice('MSFT');
```

Un exemple sans passer par un descripteur :

```
1 : use SOAP::Lite;
2 : print SOAP::Lite
3 :   ->uri('http://www.soaplite.com/Temperatures')
4 :   ->proxy('http://services.soaplite.com/temper.cgi')
5 :   ->f2c(32) # Farenheit -> Celsius
6 :   ->result;
```

Exemple RPC::XML :

```
1 : require RPC::XML;
2 : require RPC::XML::Client;
3 : $request = RPC::XML::Client->new(
4 :       'http://www.localhost.net/RPCSERV');
5 : $response = $request->send_request('some.method',
6 :                                    @arguments);
7 : print ref $response
8 :       ? join(', ', @{$response->value})
9 :       : "Error: $response";
```

Exemple XML::RPC :

```
1 : use XML::RPC;
2 : my $xmlrpc = XML::RPC->new(
3 :         'http://betty.userland.com/RPC2');
4 : my $result = $xmlrpc->call(
5 :         'examples.getStateStruct',
6 :         { state1 => 12, state2 => 28 } );
```

On ne s'intéresse qu'au côté client!

- invocation dynamiquement construite, interaction avec les DOM émis et reçus
- usage de classes de confort engendrées grâce à WSDL, méthodes bien nommées

Axis Object Model

```
1 : import org.apache.axiom.om.*;
2 : import org.apache.axis2.******;
3 :
4 : public class AXIOMClient {
5 :
6 :   private static EndpointReference targetEPR =
7 :     new EndpointReference("http://localhost:8080"
8 :         + "/axis2/services/StockQuoteService");
```

Axis Object Model

```
1 : public static OMElement
2 :         getPricePayload(String symbol) {
3 :   OMFactory fac = OMAbstractFactory.getOMFactory();
4 :   OMNamespace omNs = fac.createOMNamespace(
5 :     "http://axiom.service.quickstart.samples/xsd",
6 :     "tns");
7 :   OMElement method = fac.createOMElement(
8 :                               "getPrice", omNs);
9 :   OMElement value = fac.createOMElement(
10 :                              "symbol", omNs);
11 :  value.addChild(fac.createOMText(value, symbol));
12 :  method.addChild(value);
13 :  return method;
14 : }
```

Axis Object Model

```
1 : public static OMElement
2 :         updatePayload(String symbol, double price) {
3 :   OMFactory fac = OMAbstractFactory.getOMFactory();
4 :   OMNamespace omNs = fac.createOMNamespace(
5 :     "http://axiom.service.quickstart.samples/xsd",
6 :     "tns");
7 :   OMElement method = fac.createOMElement(
8 :                             "update", omNs);
9 :   OMElement value1 = fac.createOMElement(
10 :                            "symbol", omNs);
11 :  value1.addChild(fac.createOMText(value1, symbol));
12 :  method.addChild(value1);
13 :  OMElement value2 = fac.createOMElement(
14 :                            "price", omNs);
15 :  value2.addChild(fac.createOMText(
16 :      value2, Double.toString(price)));
17 :  method.addChild(value2);
18 :  return method;
19 : }
```

Axis Object Model

```
1 : public static void main(String[] args) throws Throwable {
2 :   OMElement getPricePayload = getPricePayload("WSO");
3 :   OMElement updatePayload =
4 :       updatePayload("WSO", 123.42);
5 :   Options options = new Options();
6 :   options.setTo(targetEPR);
7 :   options.setTransportInProtocol(
8 :       Constants.TRANSPORT_HTTP);
9 :   ServiceClient sender = new ServiceClient();
10 :  sender.setOptions(options);
11 :  sender.fireAndForget(updatePayload);
12 :  System.err.println("price␣updated");
13 :  OMElement result =
14 :      sender.sendReceive(getPricePayload);
15 :  String response =
16 :      result.getFirstElement().getText();
17 :  System.err.println("Current␣price:␣" + response);
18 : }
```

Axis Data Binding

Après génération de StockQuoteServiceStub:

```
1 : import ....adb.StockQuoteServiceStub;
2 :
3 : public class ADBClient {
4 :   public static void main(String args[]) throws Throwable {
5 :     StockQuoteServiceStub stub =
6 :       new StockQuoteServiceStub(
7 :           "http://fictif/StockQuoteService");
8 :
9 :     getPrice(stub);
10 :    update(stub);
11 :    getPrice(stub);
12 :  }
```

Axis Data Binding

Utilisation en fire and forget :

```
1 : public static void update(StockQuoteServiceStub stub) throws T
2 :   StockQuoteServiceStub.Update req =
3 :       new StockQuoteServiceStub.Update();
4 :   req.setSymbol("ABC");
5 :   req.setPrice(42.35);
6 :   stub.update(req);
7 :   System.err.println("price␣updated");
8 : }
```

Axis Data Binding

Utilisation en two way call/receive :

```
1 : public static void getPrice(StockQuoteServiceStub stub) throws
2 :   StockQuoteServiceStub.GetPrice req =
3 :       new StockQuoteServiceStub.GetPrice();
4 :   req.setSymbol("ABC");
5 :   StockQuoteServiceStub.GetPriceResponse res =
6 :       stub.getPrice(req);
7 :   System.err.println(res.get_return());
8 : }
```

Conclusion

C'était : l'approche services

- Rien de révolutionnaire, fruit d'une longue maturation
- Meilleure maîtrise de l'hétérogénéité par couplage faible
- Simplification du déploiement

La semaine prochaine : l'approche ressources