

Services Web

Chap #8

Ce chapitre fait une courte présentation des services web.

Les standards étant en perpétuelle redéfinition, il est difficile de présenter autre chose qu'un cliché instantané des technologies actuelles.

On s'attachera à montrer les aspects de communication en point à point, en soulignant les évolutions depuis les modèles CORBA et RMI.

Communications sur internet

Enjeux du “B to B”

Avec les services web, l'enjeu de la communication entre applications devient planétaire : il s'agit en effet de faire communiquer par exemple un service de réservation aérienne en un endroit donné avec un service de réservation d'hôtel à plusieurs milliers de Km de là.

Loin de la vision limitée de la communication d'entreprise, on veut pouvoir effectuer des transactions ou des échanges de données entre entreprises (“Business to Business” ou “B to B”).

Ceci est essentiel à noter pour les choix des architectures, car on souhaite ici faire communiquer de gros ensembles en couplage lâche, avec le minimum de connaissance sur l'interlocuteur, et sans pouvoir imposer aucune contrainte de service : si le site distant n'est pas disponible, il n'y a rien à faire.

Les services web sont des applications utilisant internet pour interagir dynamiquement avec d'autres programmes en s'appuyant sur des standards issus le plus souvent du W3C (World Wide Web Consortium) :

- XML (Extensible Markup Language) : pour le formatage des données.
- HTTP (Hypertext Transfer Protocol) : pour le protocole de transport.
- SOAP (Simple Object Access Protocol) : pour le protocole de communication.

XML est un standard du W3C qui permet de structurer des données sous forme textuelle à l'aide de balises, tout comme le langage HTML permet de décrire des pages web avec des balises.

Ex:

```
<personne>  
  <nom>Dupont</nom>  
  <prenom>Jean</prenom>  
</personne>
```

Les balises sont des termes entre “<>” qui définissent la syntaxe de la structure de données. La syntaxe d'un document XML peut être décrite avec XML Schema, un autre standard du W3C. Le schéma décrira le type de données utilisées (`xs:string`, `xs:double`, ...), les types de données composites, la cardinalité et la séquence des types de données.

HTTP est un protocole de communication au dessus de TCP/IP qui permet d'effectuer des requêtes sur un serveur web.

SOAP est un protocole de type RPC qui utilise XML pour sérialiser les appels de méthodes et leurs arguments, ainsi que les valeurs de retours. Avec SOAP, les documents XML transitent en général au dessus du protocole HTTP (on parle de “SOAP over HTTP”).

Communications sur internet

Intérêt d'un mode RPC sur le net

Les interfaces des services web sont décrites en WSDL (Web Service Description Language), qui joue le même rôle que l'IDL de CORBA.

- Cette description est suffisante pour utiliser le service sans connaître son implémentation, y compris à distance par internet.
- En effet, le service est couplé à un serveur web qui le met à disposition sur internet.
- L'interface WSDL est elle aussi publiée sur Internet.

Le but est de permettre à une application de communiquer sur internet, avec le service dont elle a besoin et d'échanger des données avec lui. Si le service est payant, la procédure de règlement sera comprise dans la description (WSDL) du service.

Communications sur internet

Intérêt d'un mode RPC sur le net

On remarquera que l'infrastructure de mise en oeuvre est plus lourde que les solutions telles que CORBA et RMI, puisqu'il faut utiliser un serveur web permettant de répondre à des requêtes HTTP. Par contre cette solution est incontournable dès que l'on souhaite faire transiter des informations à travers un pare-feu, car aucun des protocoles tels que RMI ou CORBA ne sont alors autorisés.

Il ne faut pas surestimer le coût de mise en oeuvre d'un serveur HTTP : la version 6 de Java en intègre un simplifié de manière native. Un serveur HTTP peut également tourner sur un équipement embarqué avec quelques classes Java, et le domaine d'application des services web pour les mobiles est potentiellement illimité.

L'utilisation de SOAP a été également rendue populaire par Microsoft, alors que .Net Remoting très proche de RMI n'a jamais été poussé en avant. Le protocole SOAP 1.2 est une recommandation du W3C depuis juin 2003.

Le nom de service vient du monde du “mainframe” où un “service” est lié à une transaction et définit une suite d’opérations cohérentes qui rend un résultat.

De manière plus générale, le concept de service réintroduit l’aspect “workflow” et la décomposition fonctionnelle dans un monde où le tout objet l’avait rendu hérétique, contrairement aux méthodes de conception type SADT des années quatre-vingt.

- On peut donc voir la notion de “service” comme un retour du fonctionnel.
- Dans le monde des services web, on conservera également l’ambiguïté entre des concepts de documents XML et des concepts d’objets.

La notion de service est souvent liée à celle de “Qualité de Service”. Si la spécification initiale de SOAP est tout à fait minimaliste, la version IBM de SOA (Service Oriented Architecture) inclut en premier lieu la qualité de service au sens des services CORBA.

Dans cette perspective, il s'agit de gérer des aspects non-fonctionnels transversaux aux différents intergiciels :

- La gestion des transactions.
- La gestion de la sécurité.
- Les annuaires, etc...

Toutefois, les spécifications récentes de SOAP permettent une interopérabilité de la qualité de service. Notamment en ce qui concerne les transactions et la sécurité. Ceci est en forte évolution et ne sera pas développé dans le cadre de ce cours.

Les services web sont l'enjeu d'une lutte commerciale féroce et d'une activité de publicité intense. De nombreuses normes s'affrontent, en particulier celles issues du W3C et du consortium OASIS.

- SOAP 1.2 est une recommandation du W3C depuis juin 2003, mais de nombreux autres standards tentent de définir une interopérabilité entre services web, y compris sur les aspects non-fonctionnels.
- On notera par exemple les travaux importants du consortium WS-I (Web Services Interoperability Organization) qui ont posé les fondements de l'interopérabilité entre services web avec le *Basic Profile* et font de même avec le *Basic Security Profile* publié en mars 2007.
- Par contre les standards de l'OASIS et du W3C sont parfois en compétition par exemple sur la sécurité avec WS-SecurityPolicy pour le W3C et WS-Security pour OASIS. L'actualité est changeante et très controversée dans tous ces domaines, et nous n'aborderons pas ces sujets en détail.

Sun n'est pas en reste avec une activité importante de normalisation des services web et de leur intégration à Java, notamment avec le JSR 181 qui permet de représenter un service à l'aide d'annotations directement dans le code.

Le langage WSDL permettant de décrire les interfaces des services web à été normalisé comme note du W3C sous la version 1.1 en mars 2001. La version 2.0 est une recommandation du 26 juin 2007.

Ce cours ne parlera pas d'une extension importante des services web décrivant leur coordination et la description de processus métier, que ce soit ebXML, BPEL ou d'autres langages de chorégraphie et orchestration de services.

De manière générale, l'utilisation des services web va reposer sur une multitude de standards dans des versions très hétérogènes (XML 1.0 avec HTTP 1.1, SOAP 1.2, WSDL 2.0, ...) et l'utilisateur devra redoubler d'attention pour mettre en oeuvre une solution réellement interopérable.

Architecture de SOAP

Trois composants fondamentaux

Tout comme on retrouve dans l'architecture CORBA l'ORB, l'IDL, le service de noms, on retrouve 3 composants équivalents dans les services web :

- **Invocation distante des services web** : SOAP - l'équivalent de l'ORB ou bus logiciel.
- **Description des services web** : WSDL - l'équivalent de l'IDL ou description de l'interface.
- **Enregistrement et découverte de services** : UDDI (Universal Description Discovery and Integration) est l'équivalent du service de noms CORBA. On notera toutefois que UDDI est relativement peu utilisé, bien que normalisé depuis 2005 sous sa version 3 par l'OASIS. Sa complexité initiale n'a pas permis un développement large d'annuaires de services. De plus, la description d'un service web à usage planétaire pose différents problèmes techniques. Mais comme l'URL du service suffit à pouvoir l'utiliser, UDDI n'est pas indispensable à l'utilisation des services web.

Architecture de SOAP

Requête HTTP et messages SOAP

Une requête HTTP est tout simplement une chaîne de caractères répondant au format HTTP.

- Exemple de requête GET :
`GET /site/page.html HTTP/1.1`
- Exemple de requête POST :
`POST /site/page.html HTTP/1.1`
`Content-Type: text/plain`
`Content-Length: 7`
`Bonjour`
- Une réponse HTTP est tout aussi simple :
`200 OK`
`Content-Type: text/plain`
`Content-Length: 12`
`Hello, World`

On va donc pouvoir en SOAP coder l'appel de méthode distant dans une requête HTTP en utilisant XML. SOAP consiste en l'envoi de messages à des objets distants avec du XML inclus dans des requêtes et réponses HTTP. On verra donc passer sur le réseau des requêtes SOAP telles que celles-ci :

Architecture de SOAP

Requête HTTP et messages SOAP

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: x
SOAPAction: "Some-URI"
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
      <symbol>DIS</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:BODY>
</SOAP-ENV:Envelope>
```

Architecture de SOAP

Requête HTTP et messages SOAP

Dans l'exemple précédent on remarquera la structure du message indiqué ci-dessous :



Architecture de SOAP

Requête HTTP et messages SOAP

La réponse va être glissée dans une enveloppe SOAP et transiter par le réseau comme ceci :

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: x
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
  <SOAP-ENV:Body>
    <m:GetLastTradePriceResponse xmlns:m="Some-URI">
      <Price>34.5</Price>
    </m: GetLastTradePriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Architecture de SOAP

Versatilité de SOAP

SOAP est un protocole ambivalent:

- On peut le considérer comme un nouvel outil RPC entre objets : les appels de méthodes et leurs arguments sont sérialisés en XML.
- On peut le considérer comme un protocole d'envoi de messages : une requête comporte un seul message, une réponse un seul message.
- On peut le considérer comme un simple protocole d'échange de documents XML, utile pour l'échange de données informatisées ("Electronic Data Interchange" ou EDI).

Ce qui différencie SOAP de XML-RPC est la présence de l'enveloppe, ainsi que la description des interfaces avec un langage spécifique, WSDL, qui sera décrit dans ce cours.

Architecture de SOAP

Versatilité de SOAP

L'enveloppe sert à préciser des informations de service, de codage des arguments, mais de plus en plus s'y ajoutent des informations d'acheminement et de qualité de service.

Certains pensent que la complexification de l'enveloppe va à terme condamner SOAP, mais le principe d'interopérabilité subsistera certainement.

SOAP dans sa version 1.2 est en fait plus complet qu'une simple RPC avec mécanisme de requête et de réponse, dans la mesure où il permet également une communication asynchrone, et de envois de message dans un seul sens.

Architecture de SOAP

Interop. des protocoles : les bindings

A leur début, les services web ont été abondamment critiqués car ils n'apportaient pas la qualité de service des autres intergiciels : HTTP est en effet un protocole sans état qui ne permet pas de gérer une session ni de gérer l'état d'un dialogue. C'est notamment pour cela qu'HTTP est considéré comme un protocole non fiable.

IBM avait lancé l'idée d'un HTTP fiable ("Reliable HTTP", RHTTP) qui n'a pas décollée. Mais HTTP est à la fois la force et la faiblesse des services web.

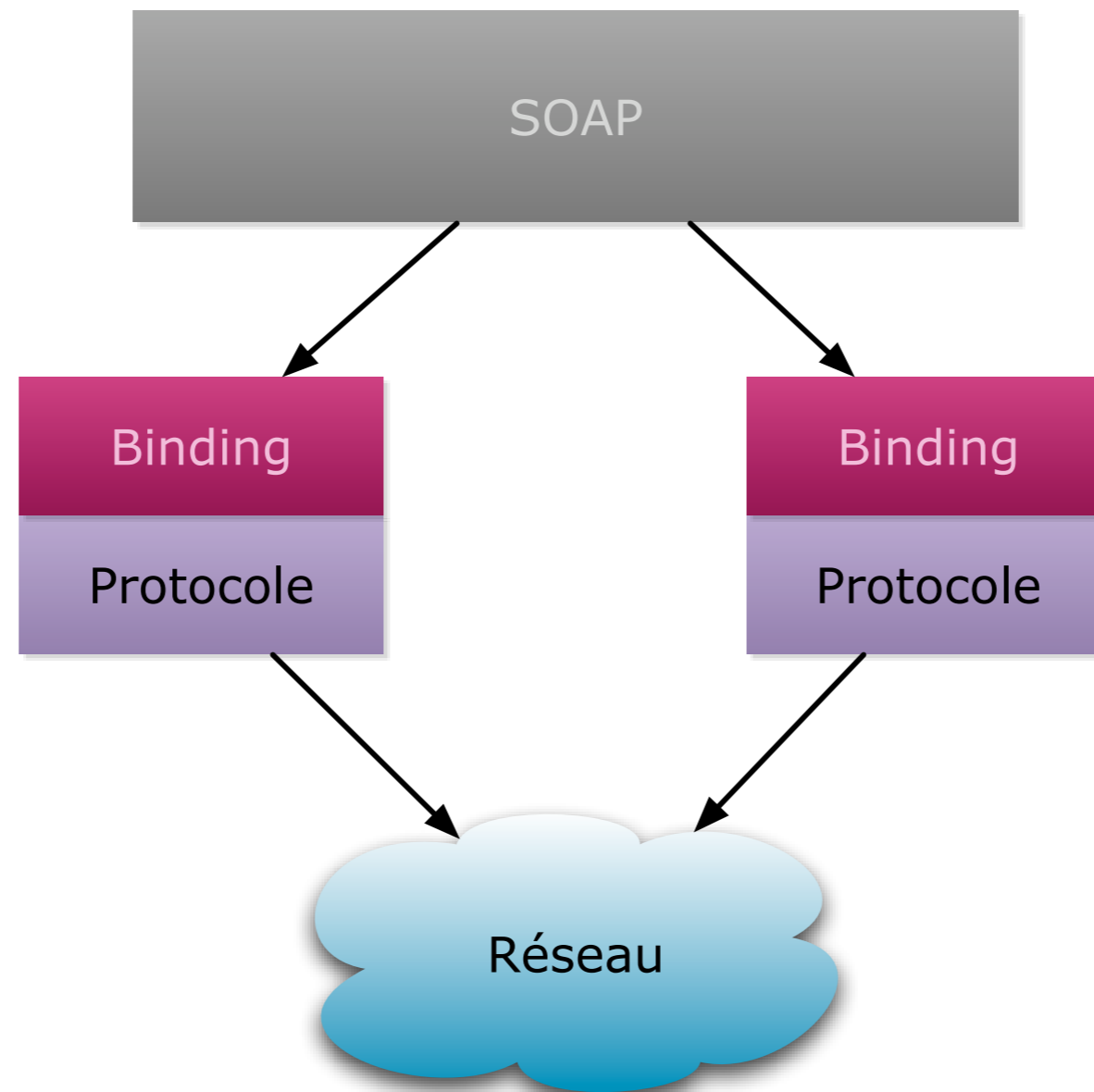
Avec la version SOAP 1.2, il est possible d'utiliser plusieurs protocoles pour implémenter une communication SOAP : on appellera cela un "binding" (une attache).

- On notera en particulier l'utilisation possible de JMS au lieu de HTTP afin d'obtenir des communication fiables avec SOAP.
- Cf. figure suivante.

Architecture de SOAP

Interop. des protocoles : les attaches

SOAP peut se lier avec différents protocoles réseau



WSDL (version 1 dans ce cours) permet de définir l'interface d'un service web tout comme un IDL CORBA. Toutefois, ce descripteur est plus riche que l'IDL car il décrit à la fois l'interface et une partie de l'implémentation (point d'accès) dans deux parties séparées.

Un fichier WSDL est un document XML qui décrit :

- **Ce que fait le service** : la liste des opérations, leur sens (requête seule ou requête-réponse), leur signature (le type des arguments). Cette partie est conceptuellement analogue à IDL.
- **Où le trouver** : le nom du service, et le point d'attache ("endpoint") pour l'invocation (le plus souvent l'URL du serveur Web hébergeur).
- **Comment l'invoquer** : les règles d'encodage des arguments, le protocole de transport utilisé.

WSDL

Un langage d'interface

Nous allons voir que WSDL, contrairement à IDL, n'a pas été conçu pour être décrit "à la main", mais généré par un outil de développement.

A partir d'une certaine taille, un fichier WSDL devient à peu près illisible pour un programmeur non averti !

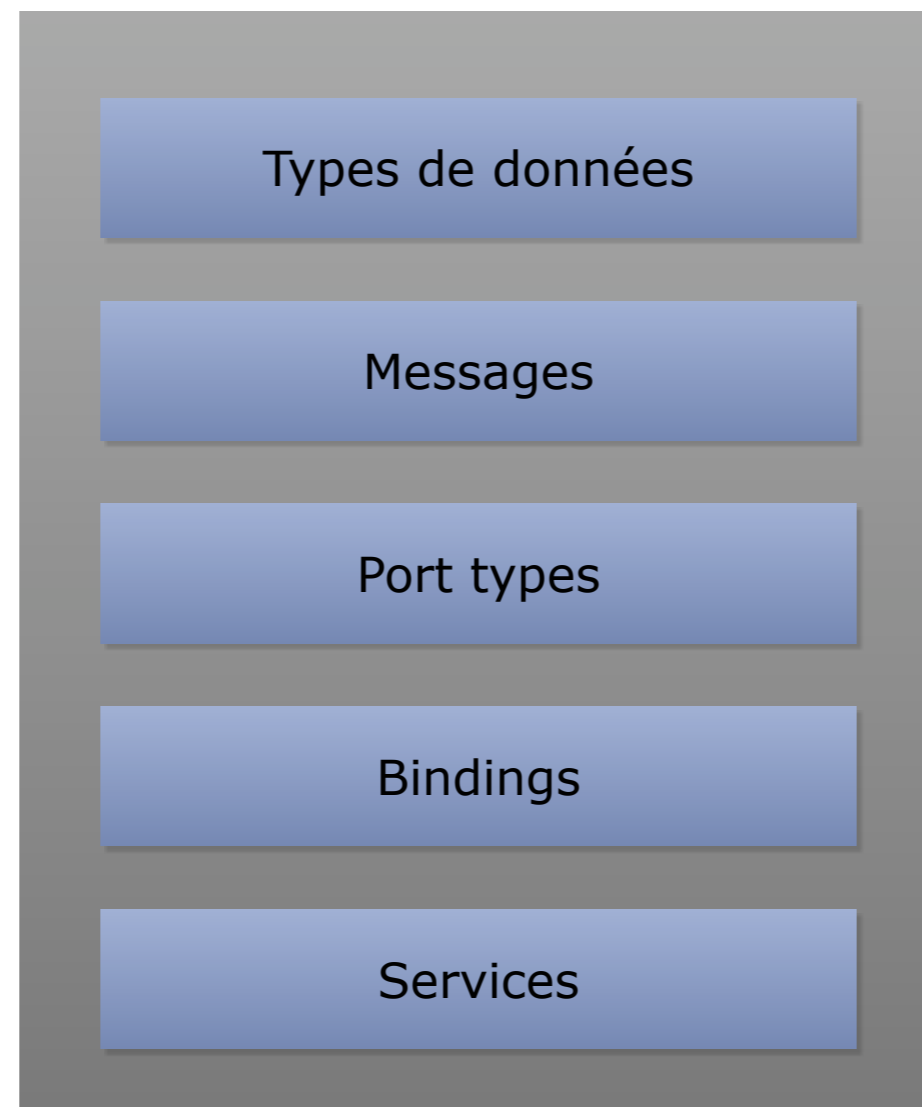
On peut toutefois débattre pour savoir s'il faut combiner au même endroit la description abstraite (type de données et signatures des méthodes) avec les aspects de déploiement (port de connexion).

WSDL

Structure d'un document WSDL

Un document WSDL peut couramment comporter plusieurs pages. Contrairement à l'IDL il sera g n ra par un outil et non manuellement.

En WSDL, un service est d fini en plusieurs parties distinctes (cf. figure suivante).



La description des types de données :

- Elle est optionnelle et permet de définir des structures de données spécifiques à l'application.
- Les types peuvent être tirés de la spécification XML Schema (comme `xsd:string`) ou être définis par l'utilisateur par composition de types simples.

Les messages :

- Ils sont unidirectionnels (sens entrant, `in`, ou sortant, `out`). Une opération RPC comportera donc en général deux messages, la requête et la réponse. Une opération `oneway` ne comportera qu'une requête.
- Un message comporte plusieurs arguments (`parts`), et chaque argument a un type défini par un schéma pouvant figurer dans le cartouche des types de données.

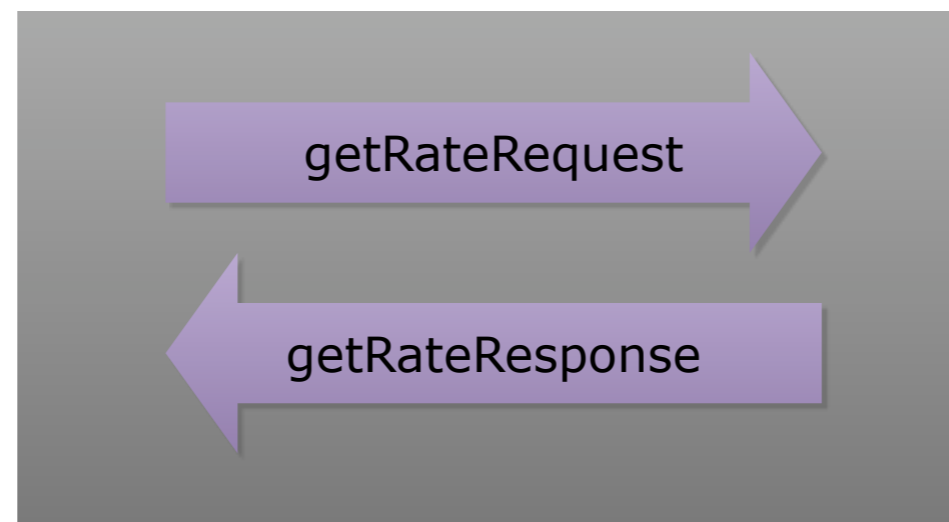
- Exemple de messages liés (requête et réponse) :

```
<message name="getRateRequest">
  <part name="country1" type="xsd:string"/>
  <part name="country2" type="xsd:string"/>
</message>
<message name="getRateResponse">
  <part name="Result" type="xsd:float"/>
</message>
```

- Ces deux messages définissent une opération qui prendra en argument d'entrée deux chaînes de caractères représentant des pays et renverra en résultat un nombre de type float représentant le cours de change.
- Le WSDL définit la signature des messages, tout comme l'IDL. Cette signature peut comporter des types prédéfinis (tirés ici de XML Schema, comme `xsd:float`). Mais il est également possible de définir des types utilisateurs en tête du WSDL.

- Voici la définition de l'opération composée des deux messages précédents (cf.figure suivante) :

```
<operation name="getRate">  
  <input message="tns:getRateRequest" />  
  <output message="tns:getRateResponse" />  
</operation>
```



operation : getRate

Le PortType :

- Le `PortType` définit l'interface abstraite du service comportant un ensemble d'opérations (cf. code suivant) (Nb: en WSDL version 2, le `PortType` est rebaptisé `Interface`).
- Chaque opération peut être soit `in-out` (appel-réponse RPC classique), `in-only` (pas de réponse ou `oneway` à la CORBA), `out-only` et `out-in` (sollicitation de la réponse).
- Chaque opération définit les messages d'entrée (`In`) ou de sortie (`Out`) ou les deux selon le mode.

```
<portType name="CurrencyExchangePortType">
  <operation name="getRate">
    <input message="tns:getRateRequest" />
    <output message="tns:getRateResponse" />
  </operation>
</portType>
```

L'attache :

- L'attache, ou `binding`, définit un lien entre le `PortType` abstrait et un service réel associé à un protocole et un format. Par exemple le binding SOAP définit le style d'encodage, l'en-tête de l'action SOAP, le namespace utilisé pour le corps XML (`targetURI`), et ainsi de suite.
- Disons sommairement qu'un `PortType` est une liste d'opération : une "interface" à la CORBA ou mieux, une facette à la CORBA 3. Un `binding` est un lien entre cette facette et un protocole (HTTP, SNMP, ...).
- En SOAP, il existe deux styles d'envoi : **RPC** et **document**. On peut également spécifier si les arguments sont codés ou littéraux.
 - Ceci définit quatre modes d'appels (cf. transparents suivants) à l'origine de beaucoup de problème d'interopérabilité dans les années passées.

- Dans l'exemple suivant, il s'agit d'un binding SOAP en mode rpc/encodé au dessus du protocole HTTP (paramètre "transport") :

```
<binding name="CurrencyExchangeBinding" type="tns:CurrencyExchangePortType">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http/">
  <operation name="getRate">
    <soap:operation soapAction="" />
    <input>
      <soap:body use="encoded" namespace="urn:xmethods-CurrencyExchange"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </input>
    <output>
      <soap:body use="encoded" namespace="urn:xmethods-CurrencyExchange"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </output>
  </operation>
</binding>
```

Le Port :

- Il définit la localisation concrète (endpoint) du service, typiquement l'URL de la servlet mettant le service à disposition.
- Dans l'exemple suivant, la servlet `http://services.xmethods.net:80/soap` (le endpoint) met à disposition le service `CurrencyExchangeService` avec l'attache décrite précédemment (référence par nom).

```
<service name="CurrencyExchangeService">
  <documentation>Renvoi le taux de change entre les deux monnaies</documentation>
  <port name="CurrencyExchangePort" binding="tns:CurrencyExchangeBinding">
    <soap:address location="http://services.xmethods.net:80/soap" />
  </port>
</service>
```

- Pour l'appel du service, il est nécessaire de connaître à la fois le point d'attache (endpoint) et le nom du service.
- Chaque `portType` n'a qu'un seul `binding`, chaque `binding` n'a qu'un seul `portType`. Inversement, **chaque service peut avoir plusieurs portTypes, chacun avec une adresse et un protocole différent.**

Comme indiqué précédemment, un binding WSDL peut avoir plusieurs styles : **RPC** ou **document**.

- Ces termes prêtent à confusion, et il ne faut pas croire que l'on ne peut faire communiquer deux applications en mode document.
- Il faut donc les prendre tels qu'ils sont, sans inférer de sémantique précise si ce n'est celle du mode de communication.

Le passage des informations peut être codé (encoded) ou littéral (literal). Ceci donne au total quatre méthodes de communication :

1. `RPC/encoded`
2. `RPC/literal`
3. `Document/encoded`
4. `Document/literal`

Comment reconnaître le style ?

- Ces différentes méthodes sont spécifiées dans le WSDL dans l'attribut `style` de `soap:binding` ainsi que dans l'attribut `use` de `soap:body`.
- L'exemple suivant présente un style `RPC/encoded` :

```
<binding name="TemperatureBinding" type="tns:TemperaturePortType">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http" />

  <operation name="getTemp">
    <soap:operation soapAction="" />
    <input>
      <soap:body use="encoded" namespace="urn:xmethods-Temperature"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </input>

  (...)

</binding>
```

- L'exemple suivant par contre est `Document/literal` (doc/lit), souvent utilisé dans le monde Microsoft :

```
<binding name="ServiceSMSSoap" type="s0:ServiceSMSSoap">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />

  (...)

  <operation name="SendSmsText">
    <soap:operation
      soapAction="http://SMSServer.dotnetISP.com/SendSmsText"
      style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
```

Exemple :

- Supposons que l'on ait une méthode : `public void myMethod(int x);`

- **Mode RPC :**

- Si l'on spécifie le style `RPC/encoded`, le message aura le contenu suivant :

```
<soap:envelope>
  <soap:body>
    <myMethod>
      <x xsi:type="xsd:int">5</x>
    </myMethod>
  </soap:body>
</soap:envelope>
```

- Plusieurs raisons pour utiliser le style `RPC/encoded` : le polymorphisme (plusieurs méthodes avec des signatures différentes), les graphes de données (utilisation de `href` en XML pour pointer d'un élément sur un autre).

- Si l'on précise au contraire le mode `RPC/literal`, le message aura le contenu suivant :

```
<soap:envelope>  
  <soap:body>  
    <myMethod>  
      <x>5</x>  
    </myMethod>  
  </soap:body>  
</soap:envelope>
```

- On voit donc que l'on économise le passage d'information de type. Par contre, il n'est plus possible de valider le contenu du message sans avoir le WSDL décrivant le type comme ceci :

```
<message name="myMethodRequest">  
  <part name="x" type="xsd:int" />  
</message>
```

- Il est à noter que le WS-I a préconisé dans son Basic Profile l'utilisation de SOAP en mode `RPC/literal` ou `Document/literal` afin d'éviter les problèmes d'interopérabilité.

- **Mode Document :**

- Si l'on précise un mode document, le message aura cette forme :

```
<soap:envelope>  
  <soap:body>  
    <xElement>5</xElement>  
  </soap:body>  
</soap:envelope>
```

- Par contre, l'élément `xElement` doit avoir été défini dans le document XML-Schema attaché au WSDL du service :

```
<types>  
  <schema>  
    <element name="xElement" type="xsd:int"/>  
  </schema>  
</types>
```

- Cette fois, il n'y a pas d'information d'encodage et il est possible de valider le contenu du message, décrit par un schéma, au prix d'une plus grande complexité du WSDL.

Les restrictions de Microsoft .Net :

- Le framework .Net de Microsoft ne supporte que le style `Wrapped Document-Literal` de WSDL :
 - Le message d'entrée a une seule partie (`part`).
 - La partie est un élément.
 - L'élément a le même nom que l'opération.
 - Le type complexe de l'élément n'a aucun attribut.

Types de données :

- Le tableau suivant précise la traduction des types de données entre WSDL et SOAP.

Spécification WSDL	Encodage SOAP
Types primitifs	
<code><element name="price" type="float"/></code>	<code><price>15.57</price></code>
<code><element name="greeting" type="xsd:string"/></code>	<code><greeting id="id1">Hello</greeting></code>
Structures	
<code><element name="Book"><complexType> <element name="author" type="xsd:string"/> <element name="title" type="xsd:string"/> </complexType></elements></code>	<code><e:Book> <author>J.R.R Tolkien</author> <title>A hobbit story</title> </e:Book></code>
Enumérations	
<code><element name="color"> <simpleType base="xsd:string"> <enumeration value="Green"/> <enumeration value="Blue"/> </simpleType></element></code>	<code><color>Blue</color></code>
Tableaux	<code><SOAP-ENC:Array id="id3" SOAP-ENC:arrayType=xsd:string[2,2]> <item>r1c1</item> <item>r1c2</item> <item>r2c1</item> <item>r2c2</item> </SOAP-EN:Array></code>

Bijections entre Java et XML

Modèle objet ou modèle de document

Une des grandes questions pour utiliser des services Web est de savoir si l'on souhaite préserver un modèle de document ou un modèle objet.

XML permet de représenter des documents, et l'on peut rester à ce niveau d'analyse en travaillant avec les concepts de documents, de noeud d'arborescence et d'attributs.

Par exemple, le listing suivant pourra être vu comme un document de racine livre, avec deux éléments, auteur et titre. L'élément titre a un attribut langue.

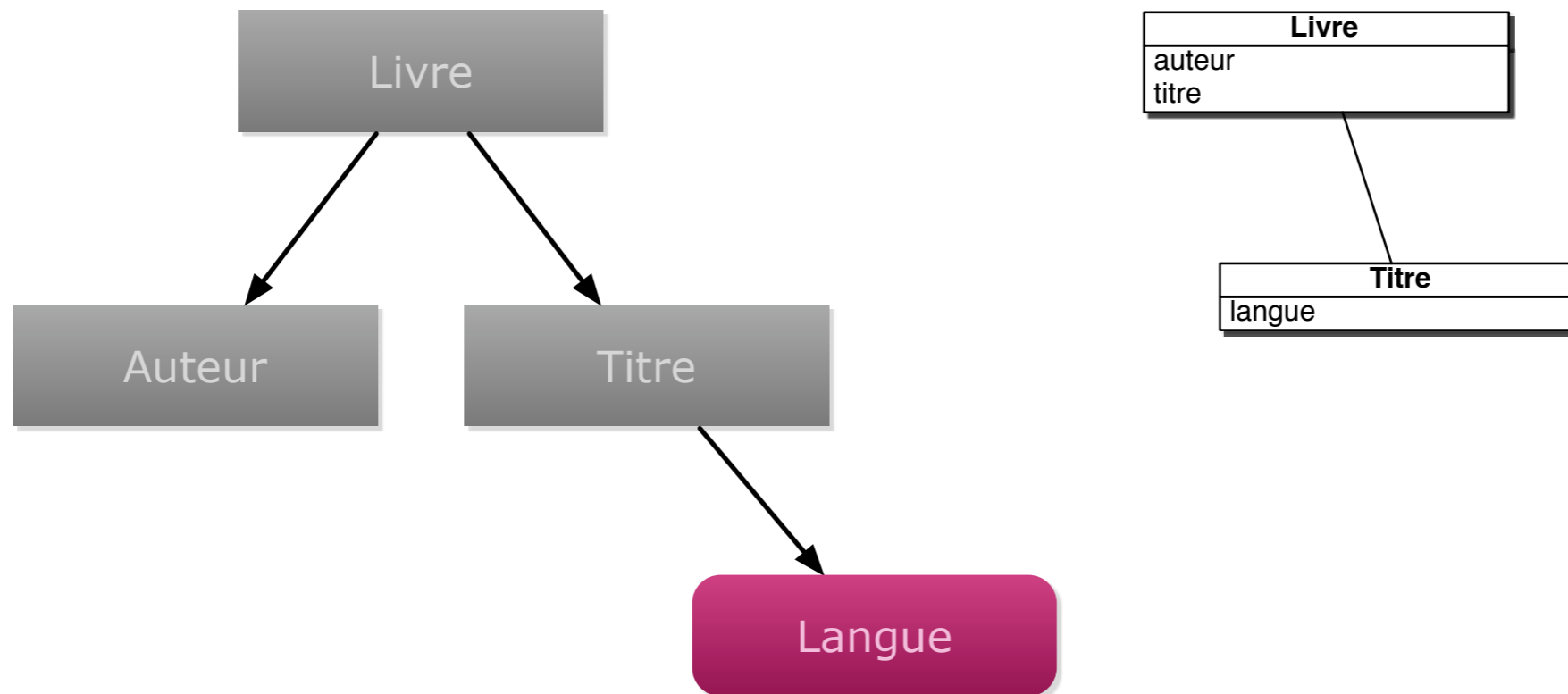
```
<livre>  
  <auteur>Goscinny</auteur>  
  <titre langue="français">Asterix en Hispanie</titre>  
</livre>
```

On peut également lui faire correspondre un objet Java de classe Livre, comportant les champs auteur et titre comme l'illustre la figure suivante.

Bijections entre Java et XML

Modèle objet ou modèle de document

Modèle XML et modèle Java :



Bijections entre Java et XML

Modèle objet ou modèle de document

Services Web | Chap #8

Doit-on créer une classer `Titre` pour rajouter l'attribut `langue` ? Un champ Java correspond-il à un élément XML ou à un attribut ?

Toutes ces questions se compliquent encore lorsque l'on prolonge l'arborescence XML.

Si le livre contient des éléments chapitres sous forme de séquence, comment traduire cela en Java ? Est-ce une `Collection` ou une `HashMap` ?

Si la séquence comporte des éléments de balise différentes, faut-il créer une nouvelle classe pour cette séquence ?

Bijections entre Java et XML

Bibliothèques Java-XML

Cette réflexion rapide montre pourquoi il y a tant de manières d'écrire des bijections entre XML et Java. On peut citer par exemple :

- **JAXB** : la bibliothèque de Sun, qui est intégrée à Java 6. Il s'agit d'un réel mapping objet entre Java et XML.
- **JAXB** : l'équivalent dans le monde Apache.
- **ATOM** : le nouveau mapping objet Apache permettant une meilleure efficacité à l'analyse grammaticale et au transfert.
- **XMLBeans** : une méthode simple à mettre en oeuvre.
- **Castor** : encore une solution en logiciel libre.
- **ADB** : Axis Databinding Framework, une alternative à JAXB dans le monde Apache.

En Java 6, la conversion Java vers XML utilise des annotations comme ci-après :

@XmlRootElement

```
class Person {
    private int _id;
    private String _name;

    public Person(){}

    public Person(int id, String name) {
        _id = id;
        _name = name;
    }
}
```

@XmlAttribute

```
public int getId(){ return _id; }
public void setId(int id) { _id = id; }
```

@XmlAttribute

```
public int getName(){ return _name; }
public void setName(int name) { _name = name; }
```

Client SOAP en Java

Choix d'implémentation

Une des difficultés avec SOAP n'est pas d'implémenter les services, mais de choisir une des multiples façons de le faire. Il existe plusieurs implémentations de SOAP en Java :

- **Sun** a la sienne propre dans Java 6 qui sera brièvement décrite dans ce cours.
- **Axis** dans le monde Apache. Notons que la version 2 a été complètement redéfinie par rapport à la version 1. Le modèle de données n'est plus imposé, mais doit être choisi parmi ATOM, JIXB, ADB et XMLBeans.
- **Xfire** sur le site de logiciel libre codehaus.

Dans les transparents suivants, nous nous intéresserons à l'implémentation fournie par Apache.

La pile (“stack”) SOAP Axis installée dans le serveur d’application web Apache Tomcat permet le **déploiement, l’hébergement et l’accès** à des services Web utilisant le protocole SOAP :

- Axis permet alors la mise au point d’applications intéropérables et réparties.
- Pour la mise en place de l’environnement on met en place le serveur Tomcat dans le répertoire `$TOMCAT_HOME`, on télécharge ensuite l’archive d’Axis (`axis.war`) que l’on copie dans `$TOMCAT_HOME/webapps`.
- On vérifie ensuite l’installation en ouvrant un navigateur Web sur l’URL :
<http://localhost:8080/axis>

Dans ce chapitre on va utiliser Axis en mode client pour accéder à un service distant permettant de récupérer le cours d’une action boursière avec un décalage de 20 minutes.

La première chose à faire est de charger le fichier WSDL représentant l'interface du service distant à l'adresse suivante :

<http://ws.cdyne.com/delayedstockquote/delayedstockquote.asmx?wsdl>

On nommera ce fichier `delayedStockQuote.wsdl`. On constate qu'il offre des opérations telles que `GetQuickStockQuote` et `GetQuickQuoteResponse`.

La requête prend les arguments suivants :

- `StockSymbol` de type `xsd:string`.
- `LicenseKey` de type `xsd:string`.

Et la réponse renvoie un paramètre de nom `GetQuickQuoteResult` et de type `xsd:decimal`.

Le WSDL fournit également les protocoles utilisables et les points d'accès au service (end-points).

Les protocoles supportés par ce service sont les suivants :

- SOAP
- SOAP 1.2
- HTTP GET
- HTTP POST

Le point d'accès est le suivant :

<http://ws.cdyne.com/delayedstockquote/delayedstockquote.asmx>

ADB signifie “Axis Databinding Framework” et permet de générer des classes Java à partir de l’analyse de schémas XML.

Par exemple, le schéma XML suivant qui comporte un élément et un type complexe générera deux classes Java, MyElement et SOAPStruct :

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://soapinterop.org/types"
  targetNamespace="http://soapinterop.org/types"
  elementFormDefault="qualified" >

  <import namespace="http://schemas.xmlsoap.org/soap/encoding/*" />
  <complexType name="SOAPStruct">
    <sequence>
      <element name="varString" type="xsd:string"/>
      <element name="varInt" type="xsd:int"/>
      <element name="varFloat" type="xsd:float"/>
    </sequence>
  </complexType>

  <element name="myElement" type="tns:SOAPStruct"/>
</schema>
```

On génère alors de manière automatique le code Java pour cette application :

- Soit en utilisant par exemple le plugin Eclipse pour la génération de code Java (voir sur le site d'Axis).
- Soit en utilisant la ligne de commande :

```
$AXIS2_HOME/bin/WSDL2Java -uri DelayedStockQuote.wsdl -p mon.package -d adb -s  
-ss -sd -ssi -o build/service
```

- On obtient alors une classe `DelayedStockQuoteStub`, qui représente une souche utilisable pour l'application cliente. Cette classe n'offre pas les méthodes escomptées comme RMI nous en donné l'habitude !
- En effet chaque requête est implémentée comme une classe interne avec un opération de requête et une opération de réponse, ainsi que des méthodes pour remplir les arguments. On perd donc de la transparence d'appel de méthodes.

Le code d'utilisation de cette souche est alors le suivant :

```
public static void main(String[] args) {
    DelayedStockQuoteStub stub;
    stub = new DelayedStockQuoteStub();
    getQuickQuote(stub);
}
```

Et la requête proprement dite :

```
public static void getQuickQuote(DelayedStockQuoteStub stub) {
    DelayedStockQuoteStub.GetQuickQuote req = new
        DelayedStockQuoteStub.GetQuickQuote(); //classe interne
    //on remplit les arguments
    req.setStockSymbol("SUNW"); //Sun
    req.setLicenseKey("0"); //Licence d'essai
    DelayedStockQuoteStub.GetQuickQuoteResponse res = stub.GetQuickQuote(req);
    System.out.println(res.getGetQuickQuoteResult());
}
```

Nous voyons dans ce code que la souche donne des guides précis pour saisir les arguments et appeler le service, mais on est loin de la transparence du mode RPC vu par exemple avec RMI.

Création d'un service Web avec Axis

Démarche de mise en oeuvre

Afin d'illustrer la simplicité de mise à disposition d'un objet Java comme service Web, nous allons considérer un exemple simple de météo. Il s'agit d'un POJO ("Plain Old Java Object"), c'est à dire un objet qui n'a besoin d'aucun héritage ni d'aucune interface non-métier, seulement les opérations et attributs nécessaires à son fonctionnement.

La mise en oeuvre comporte plusieurs étapes :

1. L'écriture de l'objet.
2. L'écriture d'un service de mise à disposition de l'objet (un peu semblable au rôle du POA en CORBA).
3. Un descriptif du service.
4. Un déploiement de l'application sous le serveur Tomcat.

Création d'un service Web avec Axis

Démarche de mise en oeuvre

On suppose que l'on a téléchargé Tomcat comme indiqué précédemment, ainsi que le fichier `axis2.war` sous `$TOMCAT_HOME/webapps`.

On vérifie l'installation en lançant Tomcat avec la commande `startup.bat` (Windows) ou `startup.sh` (*nix), puis en tapant dans un navigateur web l'URL suivante : <http://localhost:8080/axis2>

On crée par ailleurs la structure de répertoire suivante. Les fichiers seront détaillés ci-après :

Axis_pojo (*répertoire de travail*)

build.xml (*fichier ant de construction de l'application ~ make*)

src (*répertoire source*)

pojo/data/Meteo.java (*fichier Meteo.java dans le package pojo.data*)

pojo/service/ServiceMeteo.java (*fichier ServiceMeteo.java dans le package pojo.service*)

META-INF/services.xml (*fichier de déploiement dans le répertoire META-INF*)

build (*répertoire de compilation*)

Création d'un service Web avec Axis

Implémentation du service

Voici donc la classe `Meteo.java` pour implémenter un service météo. Cet objet n'a aucune contrainte particulière, ni en termes d'interface, ni en terme d'héritage. Il implémente essentiellement des accesseurs sur des variables d'instance.

```
package pojo.data;

public class Meteo {
    float temperature;
    String bulletin;
    boolean pluie;
    float hygrometrie;
    public void setTemperature(float temperature){this.temperature = temperature;}
    public float getTemperature(){return temperature;}
    public void setBulletin(String bulletin){this.bulletin = bulletin;}
    public String getBulletin(){return bulletin;}
    public void setPluie(boolean pluie){this.pluie = pluie;}
    public boolean getPluie(){return pluie;}
    public void setHygrometrie(float hygrometrie){this.hygrometrie = hygrometrie;}
    public float getHygrometrie(){return hygrometrie;}
}
```

Création d'un service Web avec Axis

Classe de service : servant

Nous avons à présent besoin d'une classe de service qui met cet objet à disposition, on parle de servant :

```
package pojo.service;
import pojo.data.Meteo;

public class ServiceMeteo {
    Meteo meteo;
    public void setMeteo(Meteo meteo){
        this.meteo = meteo;
    }

    public Meteo getMeteo(){
        return meteo;
    }
}
```

Création d'un service Web avec Axis

Fichier de déploiement

On doit ensuite créer un fichier descriptif du service `services.xml` dans le répertoire `META-INF`. Ce fichier décrit l'application `ServiceMeteo`.

```
<service name="ServiceMeteo" scope="application">
  <description>
    Service Meteo POJO
  </description>
  <messageReceivers>
    <messageReceiver
      mep="http://www.w3.org/2004/08/wsdl/in-only"
      class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
    <messageReceiver
      mep="http://www.w3.org/2004/08/wsdl/in-out"
      class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
  </messageReceivers>
  <parameter name="ServiceClass">
    pojo.service.ServiceMeteo
  </parameter>
</service>
```

Création d'un service Web avec Axis

Déploiement

Pour le déploiement, on utilise l'outil ant, sorte de "make" amélioré en écrivant le fichier build.xml suivant :

```
<project basedir="." default="generate.service">
<!-- Les différentes variables du projet -->
  <property name="service.name" value="ServiceMeteo"/>
  <property name="dest.dir" value="build"/>
  <property name="dest.dir.classes" value="{dest.dir}/{service.name}"/>
  <property name="dest.dir.lib" value="{dest.dir}/lib"/>
  <property name="axis2.home" value="c:\apps\axise2-1.2"/>
  <property name="repository.path" value="{axis2.home}/repository"/>
<!-- La génération du service -->
<target name="generate.service" depends="clean.prepare">
  <copy file="src/META-INF/services.xml"
    tofile="{dest.dir.classes}/META-INF/services.xml"
    overwrite="true" />

  <javac srcdir="src" destdir="{dest.dir.classes}"
    includes="pojo/services/**/*.pojo/data/**">
    <classpath refid="build.class.path" />
  </javac>

  <jar basedir="{dest.dir.classes}" destfile="{dest.dir}/{service.name}.aar" />
  <copy file="{dest.dir}/{service.name}.aar" tofile="{repository.path}/
    services/{service.name}.aar" overwrite="true" />
</target></project>
```

Création d'un service Web avec Axis

Déploiement

Services Web | Chap #8

Pour générer le fichier de déploiement `ServiceMeteo.aar`, il suffit de lancer la commande `ant` dans le répertoire où se trouve le fichier `build.xml`. Les fichiers `.aar` sont des archives de déploiement de services web, tout comme les fichiers `.jar`.

```
> ant
```

On copie ensuite le fichier `ServiceMeteo.aar` sous `$TOMCAT_HOME/webapps/axis2`, où `$TOMCAT` désigne le répertoire d'installation de Tomcat.

On peut alors vérifier dans un navigateur la présence du service en partant de l'URL : <http://localhost:8080/axis2/services>

Le WSDL généré est accessible en ligne à l'URL :
<http://localhost:8080/axis2/services/ServiceMeteo?wsdl>

Création d'un service Web avec Axis

Déploiement

On observe dans ce WSDL la création d'un schéma XML correspondant à l'objet météo :

```
<xs:element name="Meteo" type="ax21:Meteo" />
  <xs:complexType name="Meteo">
    <xs:sequence>
      <xs:element name="bulletin" nillable="true" type="xs:string" />
      <xs:element name="hygrometrie" type="xs:float" />
      <xs:element name="pluie" type="xs:boolean" />
      <xs:element name="temperature" type="xs:float" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

On observe également la création de trois messages : setMeteoMessage et getMeteoResponse ont des arguments, pas getMeteoMessage.

```
<wsdl:message name="setMeteoMessage">
  <wsdl:part name="part1" element="ns0:setMeteo" />
</wsdl:message>
<wsdl:message name="getMeteoMessage" />
<wsdl:message name="getMeteoResponse">
  <wsdl:part name="part1" element="ns0:getMeteoResponse" />
</wsdl:message>
```

Création d'un service Web avec Axis

Déploiement

On observe la création de deux opérations :

- setMeteo, qui utilise le message setMeteoMessage.
- getMeteo, qui utilise les messages getMeteoMessage et getMeteoResponse, un pour la requête et un pour la réponse.

```
<wsdl:operation name="setMeteo">
  <wsdl:input message="axis2:setMeteoMessage" wsaw:Action="urn:setMeteo"/>
</wsdl:operation>
<wsdl:operation name="getMeteo">
  <wsdl:input message="axis2:getMeteoMessage" wsaw:Action="urn:getMeteo"/>
  <wsdl:output message="axis2:getMeteoResponse"/>
</wsdl:operation>
```

Création d'un service Web avec Axis

Déploiement

On observe finalement donc que le mode par défaut est `document/literal` comme le montre le morceau de WSDL suivant :

```
<wsdl:operation name="setMeteo">
  <soap12:operation soapAction="urn:setMeteo" style="document"/>
  <wsdl:input>
    <soap12:body use="literal"/>
  </wsdl:input>
</wsdl:operation>
```

Création d'un service Web avec Axis

Ecriture du client RPC

On peut alors écrire le client RPC. Le point de départ est l'URL du service (endpoint), dans notre cas :

<http://localhost:8080/axis2/services/ServiceMeteo>

`localhost` est une convention réseau et désigne naturellement l'ordinateur sur lequel on travaille. Mais pour un appel à distance, il suffit de remplacer par le nom ou l'adresse IP de la machine distante.

On initialise d'abord un client RPC avec ce endpoint :

```
RPCServiceClient serviceClient = new RPCServiceClient();
Options options = serviceClient.getOptions();
EndpointReference targetEPR = new EndpointReference(
    http://localhost:8080/axis2/services/ServiceMeteo");
options.setTo(targetEPR);
```

Création d'un service Web avec Axis

Ecriture du client RPC

On peut ensuite utiliser ce client pour exécuter la méthode `setMeteo` et initialiser les valeurs du service. On remarquera que l'opération est définie par un `QName` (Qualified Name) et qu'elle utilise à l'envers le nom du package de la classe `ServiceMeteo`, `pojo.service`. Le `QName` permet d'utiliser des espaces de nommage (namespaces) afin d'éviter les collisions de noms entre plusieurs applications.

On construit les arguments comme une requête dynamique, ce qui nous permet de comprendre pourquoi cette fois on n'a pas besoin de souche spécifique.

```
private static void setMeteo(RPCServiceClient serviceClient) throws AxisFault {
    javax.xml.namespace.QName opSetWeather = new QName("http://service.pojo/xsd",
    "setMeteo");
    Meteo w = new Meteo();
    w.setTemperature((float) 25.3);
    w.setBulletin("Beau temps");
    w.setPluie(true);
    w.setHygrometrie((float) 70);
    Object[] opSetWeatherArgs = new Object[] {w};
    serviceClient.invokeRobust(opSetWeather, opSetWeatherArgs);
}
```

Création d'un service Web avec Axis

Ecriture du client RPC

De même, on peut invoquer la méthode `getMeteo` pour obtenir le bulletin :

```
private static void getMeteo(RPCServiceClient serviceClient) throws AxisFault {
    QName opGetWeather = new QName("http://service.pojo/xsd", "getMeteo");
    Object[] opGetWeatherArgs = new Object[] { };
    Class[] returnTypes = new Class[] { meteo.class };

    Object[] response = serviceClient.invokeBlocking(opGetWeather,
        opGetWeatherArgs, returnTypes);

    Meteo result = (Meteo) response[0];
    if (result == null) {
        System.out.println("Le service meteo n'est pas initialisé");
        return null;
    }
    return result;
}
```

Et finalement on peut afficher les résultats :

```
Meteo result = getMeteo(serviceClient);
System.out.println("Temperature : " + result.getTemperature());
System.out.println("Bulletin : " + result.getBulletin());
System.out.println("Pluie : " + result.getPluie());
System.out.println("Hygrometrie : " + result.getHygrometrie());
```

Création d'un service Web avec Axis

Ecriture du client RPC

Nous voyons donc qu'une des grandes limitations de l'utilisation de POJO est le fait de ne pas pouvoir initialiser l'objet du service web.

Création d'un service Web avec Java 6

Démarche de mise en oeuvre

Java 6 permet à présent de déployer facilement un service web en utilisant des annotations :

- `@WebService` : toutes les implémentations de service web doivent posséder cette annotation. Elle peut accepter des paramètres tels que `serviceName` ou `portName`.
- `@SOAPBinding` : définit le style document ou RPC.
- `@WebParam` : définit le nom d'un paramètre.
- `@WebResult` : définit l'argument de retour.
- `@OneWay` : optionnel pour les appels asynchrones.

La plate-forme met à disposition un mini serveur web avec la méthode `Endpoint.publish()`.

Création d'un service Web avec Java 6

Ecriture du client

Le client s'écrit comme un POJO avec des annotations

```
package demo;
import javax.jws.*;

@WebService(name="AFCWebServices")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class MyWebServices {
    @WebMethod
    public String hello(String nom) {
        return "Bonjour" + nom + "!";
    }

    @WebMethod(operationName="indexCorporel")
    @WebResult(name = "votre-index")
    public double indexCorporel(@WebParam(name="taille") double taille,
        @WebParam(name="poids") double poids) {
        return poids / (hauteur * hauteur) /100 * 100;
    }
}
```

Création d'un service Web avec Java 6

Publication du service

On peut utiliser une classe Java pour publier sur le serveur comme ceci :

```
public class PublierSurServeur {
    public static void main(String[] args) {
        Endpoint endpoint = Endpoint.publish("http://localhost:8080/services",
            new MyWebServices() );
        JOptionPane.showMessageDialog(null, "Eteindre le serveur");
        endpoint.stop();
    }
}
```

On peut alors vérifier le déploiement en inspectant le WSDL sous :

<http://localhost:8080/services?wsdl>

Création d'un service Web avec Java 6

Ecriture d'un client

Le client doit importer les classes générées à partir de l'interface. On utilise pour cela l'outil `wsimport`. Si le source est dans le répertoire `src`, on pourra utiliser la commande :

```
wsimport -d src -keep -p demo.gen http://localhost:8080/services?wsdl
```

Les classes générées seront dans le package `demo.gen`.

- `demo.gen.AFCWebServices.java`
- `demo.gen.MyWebServicesService.java`

On peut alors écrire le code d'appel du client :

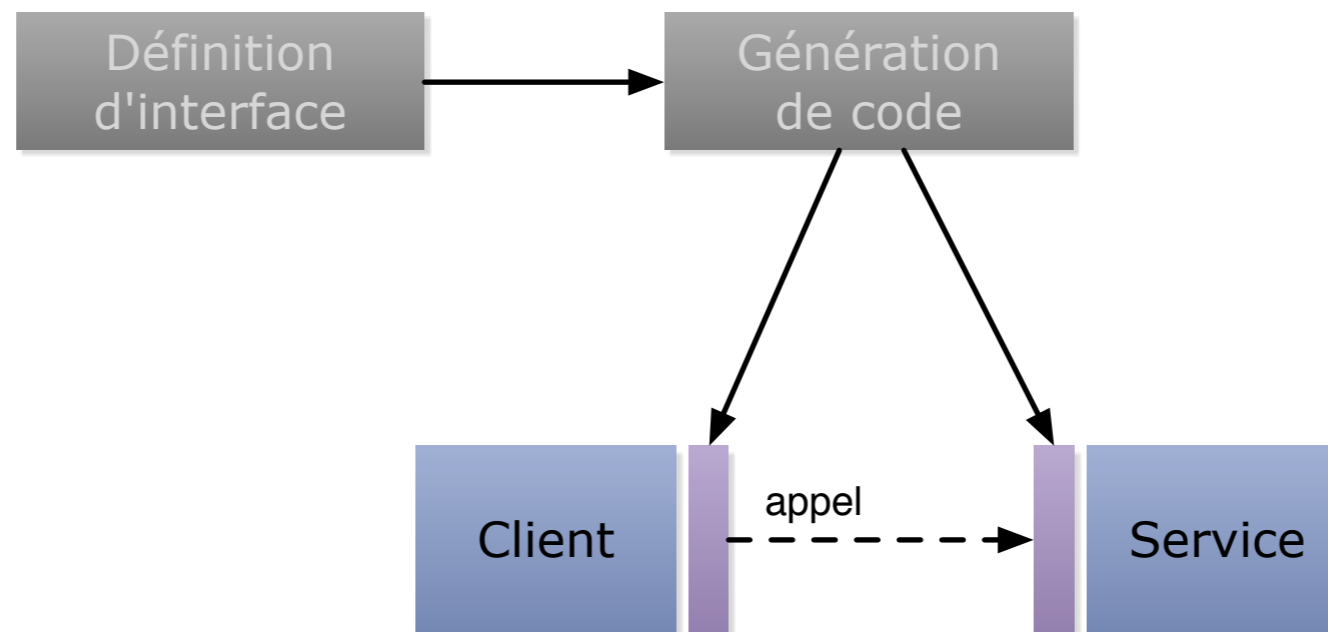
```
AFCWebServices port = new MyWebServicesService().getAFCWebServicesPort();
System.out.printf("%s Votre index est %.1f%n",
                  port.hello("Toto"),
                  port.indexCorporel(160, 60));
```

Ici l'utilisation d'une souche rend l'appel transparent, contrairement à l'exemple Axis avec ADB.

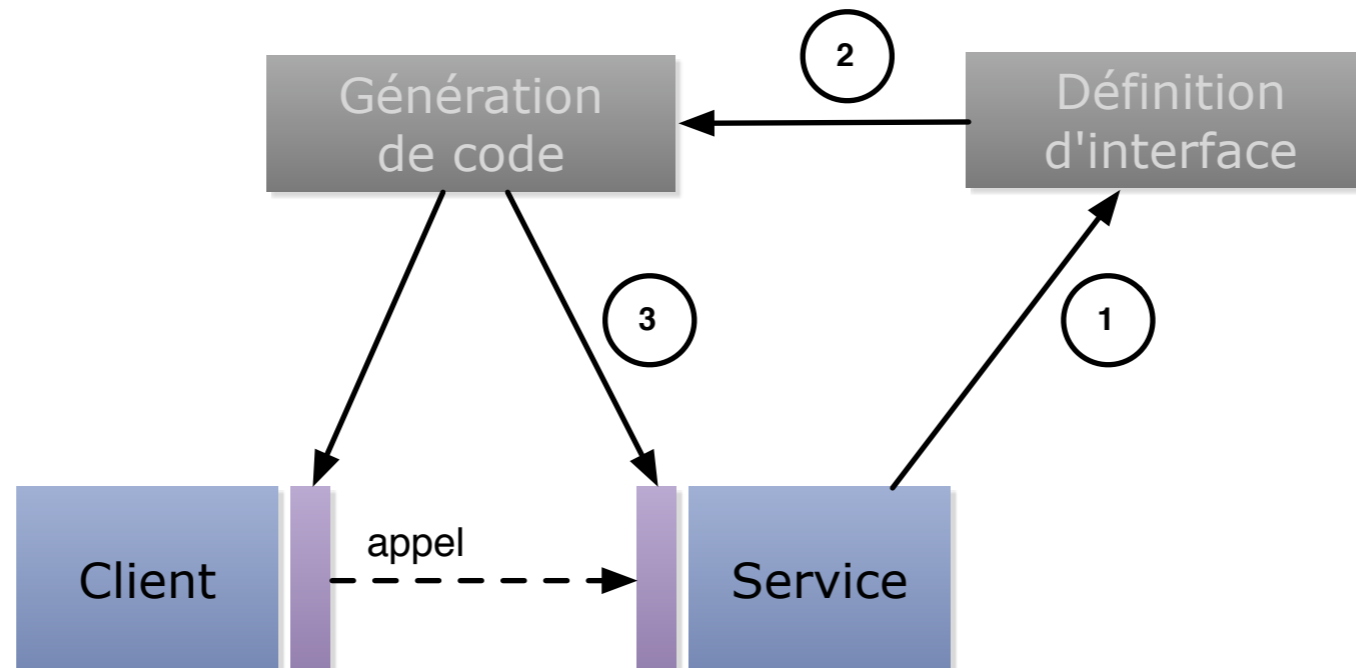
Selon que la génération du stub dépend de l'interface ou non, cela induit des architectures très différentes.

- Dans l'approche descendante, on définit d'abord l'interface, puis les outils du bus logiciel vont générer le code pour l'utilisation du bus. C'est le cas historique de CORBA (cf. figure suivante). En effet, dans cette approche l'interface précède l'implémentation et permet de générer le code souche pour le client.
- Dans l'approche montante, l'interface est extraite de l'implémentation, et n'a pas besoin d'être définie à l'avance (cf. figure suivante).

Génération de code à partir de l'interface :



Génération de l'interface à partir du code :



En CORBA, la souche est générée à partir de l'IDL, ce qui permet lors du déploiement de n'exporter que l'IDL sur la partie cliente, qui peut générer son propre code d'accès au bus.

En RMI, la souche est générée à partir de la classe compilée. Il n'est pas possible au client de générer le code de son côté à moins de disposer de l'implémentation des objets serveurs. Ceci est important pour l'utilisation des EJB (Enterprise JavaBeans), puisque l'implémentation de l'objet est générée de manière cachée par le serveur. Le conteneur EJB doit fournir des outils pour exporter la couche sans révéler l'implémentation.

En RMI, la souche peut être également téléchargée, mais cela induit des problèmes de sécurité.

En SOAP, l'interface peut être générée à partir de l'implémentation, et génère elle-même la souche. L'interface peut-être téléchargée depuis un site web sur Internet, et n'importe quel client peut générer sa propre souche pour se connecter au serveur.

En conclusion, les services Web permettent de faire communiquer des applications au travers du réseau internet en utilisant un encodage XML et le protocole SOAP.

Ces services Web sont interopérables avec d'autres langages, tels que C# de Microsoft.

Les services publient une interface d'appel dans un fichier de description d'interface WSDL publié sur le Web. Très souvent, ce fichier est généré par un outil à partir du code d'implémentation.

Un client peut générer ses propres souches pour appeler le service à partir du document WSDL publié par le web.

Les services Web sont mis à disposition dans un serveur Web, ce qui alourdit l'infrastructure, mais permet l'accès au travers de pare-feu d'entreprise.

Autour des services web gravitent de nombreux groupes de travail visant à définir des standards pour la sécurité, les transactions, etc...

XML reste un encodage très verbeux, et de nombreux travaux (XOP, ...) visent à le compresser et l'optimiser.

Qui qu'il en soit, les services web offrent une vue planétaire des échanges entre applications informatiques, et ça c'est une réelle innovation !

Fin du cours #9
