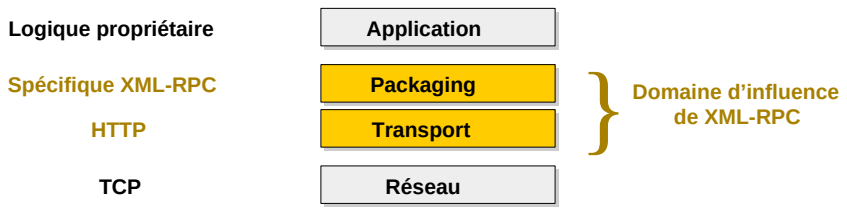
9. XML-RPC

9.1 Introduction

9.1.1 Principe

XML RPC *(Remote Procedure Call)* est un protocole de service Web pour l'exécution de procédures distantes sur plates-formes hétérogènes via l'Internet.

XML-RPC s'appuie sur **HTTP** *(Hypertext Transfer Protocol)* pour le transport et sur une syntaxe **XML** spécifique pour le packaging :



XML-RPC a été conçu au départ pour être d'une utilisation aussi simple que possible, de manière à favoriser sa dissémination en facilitant l'écriture de clients et de serveurs.

De ce fait il existe de nombreuses implémentations de **XML-RPC**, depuis la biliothèque quasi-artisanale, aux modules officiels associés à des langages comme **php**, **Perl**, **Java**, **python**, **.NET**, ou même directement intégrées à des serveurs comme **Apache**.

9.1.2 Fiche d'identité

> Références

☞ Site de référence : www.xmlrpc.com
[http://www.xmlrpc.com/]

Les spécifications de **XML-RPC** ont été initialement développées en 1999 par *Dave Winer* de la société **Userland Software**, puis subi des modifications mineures dont la dernière date de juin 2003.

UserLand Software est un éditeur de logiciels pour la publication de sites Web et de *weblogs*. Leur produit-phare est **Manila**.

☞ UserLand Software
[http://www.userland.com/]
☞ Manila
[http://manila.userland.com]

> Déclaration de type de document

Officiellement **XML-RPC** ne dispose pas d'un **URI** public, ni d'une **DTD**, et encore moins d'un schéma. Un document **XML-RPC** ne comporte donc généralement pas de déclaration *"DOCTYPE"* et ne peut être directement validé par les outils **XML** existants *(ce qui n'empêche pas de contrôler sa cohérence au niveau applicatif)*.

Il existe toutefois des **DTD** et des schémas *(parfois partiels)* mis à disposition par la communauté.

☞ Exemple de DTD pour XML-RPC
[http://code.haskell.org/haxr/xml-rpc.dtd]
☞ Page de cours sur XML-RPC, avec DTD et schéma
[http://www.cafeconleche.org/books/xmljava/chapters/ch02s05.html]

> Espace de noms

XML-RPC ne gère pas les espaces de noms **XML**. Un document **XML-RPC** ne peut donc pas mêler des éléments provenant d'applications différentes, et tous les éléments d'un tel document doivent appartenir à l'espace de noms par défaut.

> Remarques

L'absence de DTD, de schéma, ou d'espace de noms ne sont pas les seules limitations de **XML-RPC**, dont les spécifications ne sont pas issues d'un organisme comme le **W3C** ou **OASIS** par exemple, mais ont été développées par une personne seule, sans vrai suivi ultérieur pour en améliorer la robustesse.

Nous pointerons par la suite quelques erreurs de conception de **XML-RPC**, non dans un but critique *(car XML-RPC est parfaitement utilisable - et utilisé - tel quel)*, mais dans un but pédagogique afin de souligner la nécessité d'un travail de groupe en comité d'experts pour développer des spécifications robustes, interopérables et adaptées à leur écosystème.

9.1.3 Exemple d'échange

XML-RPC s'appuie sur le protocole **HTTP** pour l'échange de documents **XML**, dont le contenu décrit la requête et sa réponse.

Voici un exemple de requête, qui s'adresse à un service renvoyant le nom d'un département français lorsqu'on l'interroge en fournissant le numéro :

> Requête

```
POST /~muller/cours/xml/appl/xmlrpc-server.php HTTP/1.0
Host: tic01.tic.ec-lyon.fr
User-Agent: xmlrpc-client.php
Content-Type: text/xml
Content-Length: 153
Connection: Close

<?xml version="1.0"?>
<methodCall>
 <methodName>NomDepartement</methodName>
 <params>
  <param>
   <value>69</value>
  </param>
 </params>
</methodCall>
```

> Réponse

```
HTTP/1.1 200 OK
Date: Tue, 14 Sep 2004 21:34:38 GMT
Server: Apache/1.3.28
X-Powered-By: PHP/4.3.3
Connection: close
Content-Length: 170
Content-Type: text/xml

<?xml version="1.0"?>
<methodResponse>
 <params>
  <param>
   <value>
    <string>Rh&#xF4;ne</string>
   </value>
  </param>
 </params>
</methodResponse>
```

9.2 Format de la requête

9.2.1 Entêtes HTTP

Voici l'entête **HTTP** d'une requête **XML-RPC** typique :

```
POST /~muller/cours/xml/appl/xmlrpc-server.php HTTP/1.0
Host: tic01.tic.ec-lyon.fr
User-Agent: xmlrpc-client.php
Content-Type: text/xml
Content-Length: 153
Connection: Close
```

> Requête HTTP

```
POST /~muller/cours/xml/appl/xmlrpc-server.php HTTP/1.0
```

Une requête **XML-RPC** s'effectue obligatoirement avec la méthode **POST**.

L'**URI** de la requête est non significative pour **XML-RPC**. En ce qui concerne le service Web, cette **URI** pourrait aussi bien être vide et prendre la valeur *"/"*.

Toutefois, lorsque le serveur n'est pas uniquement un serveur **XML-RPC** et qu'il traite donc également d'autres requêtes **HTTP**, il est toléré que l'**URI** de la requête soit non vide, de manière à permettre au serveur de localiser précisement le service demandé *(cf. exemple ci-dessus)*.

N.B. La version **HTTP** de la requête est obligatoirement **HTTP/1.0** *(cf. justification dans la forme de la réponse)*.

> Directives HTTP

```
Host: tic01.tic.ec-lyon.fr
User-Agent: xmlrpc-client.php
Content-Type: text/xml
Content-Length: 153
Connection: Close
```

Les directives **Host** *(nom du serveur virtuel)*, **User-Agent** *(identification du client)*, **Content-Type** *(type du corps de la requête)* et **Content-Length** *(taille du corps en octets)* sont obligatoires.

La directive **Content-Type** prend obligatoirement la valeur *"text/xml"*.

La directive **Content-Length** est obligatoire, la valeur associée est obligatoirement correcte et doit correspondre au nombre d'octets du corps de la requête.

XML-RPC ne précise pas explicitement si d'autres directives *(comme ici Connection: close)* sont autorisées ou non. Toutefois, l'usage veut que de telles directives soient acceptées par les serveurs *(ce qui va bien dans le sens de la philosophie HTTP)*.

N.B. Certaines implémentations utilisent cette possibilité pour compléter **XML-RPC** avec l'un des mécanismes d'authentification reconnus par **HTTP** *(Basic ou Digest)*, et pour gérer des sessions à l'aide de *"cookies"*.

9.2.2 Corps de la requête

Le corps de la requête **XML-RPC** contient un document **XML** bien formé. L'élément racine doit être *"methodCall"* :

```
<?xml version="1.0"?>
<methodCall>
 <methodName>NomDepartement</methodName>
 ...
</methodCall>
```

Les spécifications ne précisent pas si la **déclaration XML** *(1ère ligne du document)* est obligatoire ou non, ni quels sont les codages de caractères autorisés.

Pour une interopérabilité maximale, il est donc conseillé pour un client de faire figurer la déclaration **XML** et de n'employer que les codages **UTF-8** ou **UTF-16**.

Un serveur par contre, pourra autoriser un document sans déclaration **XML** *(non obligatoire en XML)* et accepter tous les codages qui lui conviennent.

> Procédure

L'élément *"methodCall"* doit contenir un élément *"methodName"* dont le contenu est une chaîne de caractères qui donne le nom de la procédure appelée.

Le nom de la procédure doit être composé uniquement de caractères alphanumériques parmi *[a-z A-Z 0-9]* et des 4 caractères de ponctuation *[_ . : /]*

Il appartient entièrement au service considéré d'interpréter le nom de la procédure, qui peut effectivement correspondre à un sous-programme qu'il s'agit d'appeler *(en Java, Perl, Python, php...)* ou bien à tout autre chose comme le nom d'une table dans une base de données ou bien le nom d'un fichier à analyser.

> Paramètres

Si la procédure appelée comporte des paramètres, alors l'élément *"methodName"* doit être suivi d'un unique élément *"params"*, qui contient autant d'éléments *"param"* qu'il y a de paramètres.

```
<methodCall>
 <methodName>NomDepartement</methodName>
 <params>
  <param>...</param>
   ...
 </params>
</methodCall>
```

Chacun des éléments *"param"* contient à son tour un unique élément *"value"* dont le contenu dépend du type du paramètre considéré :

```
  <param>
   <value>Raymond Deubaze</value>
  </param>
```

N.B. XML-RPC reconnait divers types de données, scalaires, structures ou tableaux *(array)*, qui seront abordés dans la suite du cours.

9.2.3 Paramètres

D'après les spécifications, on peut provisoirement résumer la syntaxe d'une requête **XML-RPC** à l'aide de l'extrait de **DTD** suivant :

```
<!ELEMENT methodCall (methodName, params?)>
<!ELEMENT methodName (#PCDATA)>
<!ELEMENT params     (param+)>
<!ELEMENT param      (value)>
```

Ces quelques règles font clairement apparaitre que, conformément à une interprétation raisonnable des spécifications, l'élément *"params"* peut être omis *(procédure sans paramètres)*, et que s'il existe, il contient au moins un élément *"param"*.

Exemple d'appel de procédure sans paramètres :

```
<methodCall>
 <methodName>ListeDepartements</methodName>
</methodCall>
```

> Problème d'interopérabilité

Certains clients procèdent de la manière qui vient d'être décrite.

Il se trouve malheureusement que certaines implémentations de serveurs ont interprété les spécifications de manière différente. Selon ces implémentations, l'élément *"params"* serait obligatoire, quitte à être vide :

```
<methodCall>
 <methodName>ListeDepartements</methodName>
 <params/>
</methodCall>
```

Lorsqu'un client envoie une requête sans *"params"* à un serveur qui considère que cet élément est obligatoire, nous sommes en présence d'un problème d'interopérabilité.

> Solution au problème

Il se trouve qu'un élément *"params"* vide ne semble poser problème à aucun serveur connu.

En cas d'absence de paramètres, l'interopérabilité maximale est alors obtenue pour un client en envoyant un élément *"params"* vide, et pour un serveur en acceptant indifféremment les deux constructions *(params absent ou vide)*.

Cette démarche semble aujourd'hui avoir été adoptée par la plupart des implémentations *(python, php, ...)*. Elle est résumée par les deux **DTD** ci-dessous :

Client :

```
<!ELEMENT methodCall (methodName, params)>
<!ELEMENT methodName (#PCDATA)>
<!ELEMENT params     (param*)>
<!ELEMENT param      (value)>
```

Serveur :

```
<!ELEMENT methodCall (methodName, params?)>
<!ELEMENT methodName (#PCDATA)>
<!ELEMENT params     (param*)>
<!ELEMENT param      (value)>
```

N.B. La **DTD** couramment trouvée sur Internet est la version *"Client"* ci-dessus. Implémentée côté serveur elle conduit au problème qui vient d'être évoqué.

9.2.4 Schéma d'une requête

Construisons un **schéma XML** qui traduit la syntaxe d'une requête **XML-RPC**.

L'élément *"methodCall"* contient un élément *"methodName"* suivi par un élément *"params"* optionnel :

```
<xs:element name="methodCall">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="methodName" type="T_NAME"/>
   <xs:element name="params" minOccurs="0" type="T_PARAMS"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

L'élément *"methodName"* contient une chaîne de caractères non vide, dont les caractères autorisés sont explicitement cités :

```
<xs:simpleType name="T_NAME">
 <xs:restriction base="xs:string">
  <xs:pattern value="[A-Za-z0-9/.:_]+"/>
 </xs:restriction>
</xs:simpleType>
```

N.B. La chaîne est non normalisée *(le type hérite de "xs:string")*, ce qui signifie que les espaces sont interdits, y compris en tête et en fin de chaîne. Ceci signifie que la requête ci-dessous est incorrecte :

```
<methodCall>
 <methodName>
  getInfo
 </methodName>
</methodCall>
```

L'élément *"params"* contient une liste éventuellement vide d'éléments *"param"* :

```
<xs:complexType name="T_PARAMS">
 <xs:sequence>
  <xs:element name="param" minOccurs="0" maxOccurs="unbounded" type="T_PARAM">
 </xs:sequence>
</xs:complexType>
```

L'élément *"param"* contient un unique élément *"value"* :

```
<xs:complexType name="T_PARAM">
 <xs:sequence>
  <xs:element name="value" type="T_VALUE"/>
 </xs:sequence>
</xs:complexType>
```

Le contenu de l'élément *"value"* sera décrit par ailleurs.

☞ Voir l'extrait de schéma complet
[http://dmolinarius.github.io/demofiles/mod-84/xrpc/request/request.xsd.html]

9.3 Valeurs scalaires

9.3.1 Nombres entiers

Les nombres entiers **XML-RPC** sont représentés par un élément *"int"* ou *"i4" (notations équivalentes)*, dont le contenu est un entier signé codé sur 32 bits.

Vue la définition ci-dessus, les valeurs autorisées vont par conséquent de -2147483648 à +2147483647 inclus.

Exemple de valeurs entières :

```
<value><int>69</int></value>
<value><i4>2005</i4></value>
```

> Schéma XML

L'ensemble des valeurs autorisées pour les entiers **XML-RPC**, correspond à l'espace des valeurs du type prédéfini *"xs:int"*. Toutefois, **xs:int** et normalisé alors que les spécifications **XML-RPC** interdisent les espaces de tête et de fin.

En toute rigueur, si l'on veut absolument représenter exactement l'espace lexical autorisé pour les entiers **XML-RPC**, il faut définir *(non sans malice)* un type dérivé de *"xs:string"*. Cette opération sera détaillée à titre pédagogique dans le prochain paragraphe.

9.3.2 Vous avez dit normalisé ?

Les spécifications de **XML-RPC** précisent explicitement *(commentaires du 21/1/99)* que les espaces de tête et de fin sont interdits dans l'expression des valeurs scalaires *(comme int par exemple)*.

Le **schéma XML** qui permet de représenter les entiers n'est pas simple à imaginer.

> Dérivation à partir de xs:int

L'espace des valeurs des entiers **XML-RPC** correspond au type prédéfini *"xs:int"*. Toutefois, ce type est normalisé, c'est à dire qu'il accepte les espaces de tête et de fin.

Afin de restreindre l'espace lexical de *"xs:int"*, on pourrait être tenté d'interdire les espaces grâce à la facette *"xs:pattern" (qui agit bien sur l'espace lexical)* :

```
<xs:simpleType>
 <xs:restriction base="xs:int">
  <xs:pattern value="\S+"/> <!-- limité aux non-espaces -->
 </xs:restriction>
</xs:simpleType>
```

Toutefois, cette méthode ne fonctionne pas, car la facette *"xs:pattern"* travaille sur l'espace lexical **après** normalisation.

> Dérivation à partir de xs:string

La seule solution consiste à dériver depuis *"xs:string" (non normalisé)*, en n'autorisant que les expressions qui correspondent à des entiers :

```
<xs:simpleType>
 <xs:restriction base="xs:string">
  <xs:pattern value="[+\-]?[0-9]+"/>
 </xs:restriction>
</xs:simpleType>
```

Malheureusement, cette méthode n'est pas satisfaisante non plus, car elle ne limite pas l'espace des valeurs pour correspondre exactement à celui des entiers 32 bits.

> Solution

La seule solution vraiment satisfaisante est assez laborieuse, puisqu'elle consiste effectivement à construire le type désiré à partir de *"xs:string"*, mais en listant l'ensemble des combinaisons autorisées.

La technique retenue ici revient à autoriser d'abord tous les nombres à 9 chiffres significatifs *(les spécifications précisent que les zéros initiaux sont ignorés)*, puis à lister au fur et à mesure toutes les combinaisons autorisées de nombres à 10 chiffres.

Cette approche est appliquée au sein du modèle suivant dont l'espace lexical correspond effectivement à celui des entiers **XML-RPC** *(les sauts de ligne ont été ajoutés uniquement pour faciliter la lecture)* :

```
<xs:simpleType name="T_INT">
 <xs:restriction base="xs:string">
  <xs:pattern value="[+\-]?(0+|0*([1-9][0-9]{0,8}|1[0-9]{9}|2(0[0-9]{8}|
                     1([0-3][0-9]{7}|4([0-6][0-9]{6}|7([0-3][0-9]{5}|4(
                     [0-7][0-9]{4}|8([0-2][0-9]{3}|3([0-5][0-9]{2}|6([0-3]
                     [0-9]|4[0-7]))))))))))|-0*2147483648"/>
 </xs:restriction>
</xs:simpleType>
```

☞ Voir un extrait de schéma détaillant le raisonnement employé
[http://dmolinarius.github.io/demofiles/mod-84/xrpc/values/int.xsd.html]

9.3.3 Booléens

Les booléens **XML-RPC** sont représentés par l'élément *"boolean"*.

Les seules valeurs autorisées sont *"0"* et *"1"* .

Exemple de booléens :

```
<value><boolean>0</boolean></value>
<value><boolean>1</boolean></value>
```

> Schéma XML

Comme les spécifications interdisent les espaces de tête et de fin, il faut, là encore, recourir au type *"xs:string"* :

```
<xs:simpleType name="T_BOOLEAN">
 <xs:restriction base="xs:string">
  <xs:pattern value="0|1"/>
 </xs:restriction>
</xs:simpleType>
```

9.3.4 Nombres à virgule flottante

XML-RPC représente les nombres à virgule flottante par un élément *"double"*, dont le contenu est l'expression lexicale d'un nombre *"signé double précision"*.

Les spécifications *(commentaires du 21/1/99)* précisent toutefois que :

- il n'est pas possible de représenter *"NaN" (Not a Number)*, ni les infinis positif ou négatif,
- la seule représentation lexicale autorisée prend la forme d'un nombre décimal *(pas d'exposant autorisé)*,
- le nombre de chiffres est illimité *(!)*,
- les valeurs maximales et minimales admises sont *"implementation dependant"*.

Exemple de valeurs à virgule flottante :

```
<value><double>3.141592</double></value>
<value><double>-273.29</double></value>
```

> Schéma XML

L'expression *"signé double précision"* tendrait de prime abord à indiquer un espace des valeurs correspondant à celui du type prédéfini *"xs:double"*. Pourtant, les commentaires indiquent qu'il s'agit plutôt de *"xs:decimal"*.

Cependant, les espaces étant toujours interdits, il faut encore une fois se rabattre sur *"xs:string"* :

```
<xs:simpleType name="T_DOUBLE">
 <xs:restriction base="xs:string">
  <xs:pattern value="[+\-]?[0-9]+\.[0-9]+"/>
 </xs:restriction>
</xs:simpleType>
```

9.3.5 Dates et heure

Les spécifications **XML-RPC** prévoient également des valeurs de type date, dont l'élément nommé *"dateTime.iso8601"* fait référence au standard **IS0 8601**.

Le standard **ISO 8601** est plutôt complexe, et prévoit de nombreux formats de date, d'heure, et d'intervalles de temps.

Toutefois, et malgré l'abîme de flou laissé par les spécifications *(aucun commentaire, 1 exemple !)*, les usages en cours au sein de la communauté tendent à indiquer que le seul format reconnu est `CCYYMMDDTHH:MM:SS`.

Ce format comporte 4 digits pour l'année, immédiatement suivis par le mois sur 2 digits, le jour sur 2 digits, la caratère *"T"* obligatoire, puis l'heure, les minutes et les secondes, chaque fois sur deux digits, séparées par des caractères *":"*.

Exemple de date :

```
<value><dateTime.iso8601>20050115T20:18:17</dateTime.iso8601></value>
```

La valeur ci-dessus signifie *"15 janvier 2005, 20 heures 18 minutes et 17 secondes"*.

Aucun autre format ne semble devoir être autorisé.

> Schéma XML

Bien que le seul format possible soit assez rigide, le schéma n'est encore une fois pas simple à définir.

En effet, et bien que **XML Schéma** dispose de types prédéfinis relatifs aux dates **IS0 8601**, l'interdiction des espaces de tête et de fin nécessite à nouveau le recours à *"xs:string"*, ce qui impose de contrôler manuellement le nombre de jours de chacun des mois, ainsi que les années bissextiles.

Une fois encore, le raisonnement consiste à juxtaposer toutes les solutions autorisées...

☞ Voir un extrait de schéma détaillant le raisonnement employé
[http://dmolinarius.github.io/demofiles/mod-84/xrpc/values/date.xsd.html]

Cette approche est appliquée au sein du modèle suivant *(les sauts de ligne ont été ajoutés uniquement pour faciliter la lecture)* :

```
<xs:simpleType name="T_DATE">
 <xs:restriction base="xs:string">
  <xs:pattern value="([0-9]{4}((0(1|3|5|7|8)|1(0|2))(0[1-9]|[1-2][0-9]|
                     3[0-1])|(0(4|6|9)|11)(0[1-9]|[1-2][0-9]|30)|02(0[1-9]
                     |1[0-9]|2[0-8]))|(([0-9]{2}(0(4|8)|(1|3|5|7|9)(2|4|6)
                     |(2|4|6|8)(0|4|8))|[0-9]000)0229))
                     T(((0|1)[0-9])|(2[0-3])):[0-5][0-9]:[0-5][0-9]"/>
 </xs:restriction>
</xs:simpleType>
```

9.3.6 Chaînes de caractères

Les chaînes de caractère **XML-RPC** sont représentées par l'élément *"string"*.

Tous les caractères des chaînes **XML-RPC** sont significatifs *(espaces compris)*. Conformément aux spécifications de **XML**, les caractères *"<"* et *"&"* doivent évidemment être remplacés par les appels d'entités *"<"* et *"&"*.

La première version des spécifications **XML-RPC** restreignait les chaînes aux caractères **ASCII**, mais cet aspect a été modifié par la suite en autorisant tous les caractères *(?)*

La liste exacte des caractères autorisés par les spécifications **XML-RPC** a donné lieu à de nombreuses spéculations sur Internet. Il en découle qu'un serveur peut toujours accepter des caractères autres que **US-ASCII**, mais qu'un client ne pourra en profiter que s'il en est averti *(bien entendu, il n'existe aucun mécanisme de négociation des possibilités du serveur)*.

Exemple de valeur chaîne :

```
<value>
 <string>
  Any technology distinguishable from magic is insufficiently advanced
 </string>
</value>
```

Lorsqu'un élément *"value"* contient directement du texte *(sans indication de type)*, alors **XML-RPC** considère qu'il s'agit d'une chaîne de caractères :

```
<value>
 This is magic !
</value>
```

Afin d'optimiser l'interopérabilité il est conseillé d'accepter cette dernière syntaxe côté serveur, mais de ne pas l'employer côté client...

> Schéma XML

En toute rigueur les chaînes **XML-RPC** correspondent au type prédéfini *"xs:string"* limité aux caractères **ASCII** plus les espaces :

```
<xs:simpleType name="T_STRING">
 <xs:restriction base="xs:string">
  <xs:pattern value="(&#x9;|&#xD;|&#xA;|[&#x20;-&#x7F;])*"/>
 </xs:restriction>
</xs:simpleType>
```

D'un autre côté, il est impossible d'exprimer dans un **schéma XML** la possibilité de ne pas faire figurer la balise *"<string>"* lorsqu'un élément *"value"* contient une chaîne.

Si l'on tient vraiment à autoriser ce comportement, la seule solution consiste à déclarer que l'élément *"value"* accepte un contenu mixte, ce qui conduit à valider de nombreux documents non conformes à **XML-RPC**...

9.3.7 Données binaires

Les documents **XML** sont des documents texte. Une solution pour intégrer des données binaires consiste à les encoder de manière à obtenir un flux de caractères autorisés.

A cet effet, **XML-RPC** prévoit explicitement les valeurs codées en base 64, spécifiées grâce à l'élément *"base64"*.

Exemple de chaîne encodée en base 64 :

```
<value>
 <base64>RXhlbXBsZSBkZSBjaGHubmUgY29k6WUgZW4gYmFzZSA2NA==</base64>
</value>
```

> Schéma XML

Les valeurs *"base64"* de **XML-RPC** correspondent au type prédéfini *"xs:base64Binary"*.

Ce type est un type normalisé. Toutefois, cet aspect n'est pas rédhibitoire dans la mesure où la plupart des implémentations de l'algorithme base 64 autorisent *(et ignorent)* les espaces et les retours à la ligne au sein de la chaîne encodée.

9.4 Types composés

9.4.1 Tableaux

Outre des données scalaires, l'élément *"value"* de **XML-RPC** peut contenir un élément *"array"*, formé d'un élément *"data"*, qui contient lui-même des éléments *"value"*.

Exemple de tableau :

```
<value>
 <array>
  <data>
   <value><int>69</int></value>
   <value><base64>Umj0bmU=</base64></value>
   <value><string>Lyon</string></value>
   <value><base64>Umj0bmUtQWxwZXM=</base64></value>
  </data>
 </array>
</value>
```

L'exemple ci-dessus correspond à un tableau qui comprend quatre valeurs *(noter que "base64" a été ici utilisé pour coder les chaînes contenant des accents)*. Remarquer que les valeurs contenues dans un tableau ne sont pas forcément toutes du même type.

Un tableau peut contenir des valeurs qui contiennent elles-mêmes des tableaux ou des structures.

> Schéma XML

Conformément à la définition ci-dessus, le schéma d'une valeur du type *"array"* est le suivant :

```
<xs:complexType name="T_ARRAY">
 <xs:sequence>
  <xs:element name="data" type="T_ARRAY_DATA"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="T_ARRAY_DATA">
 <xs:sequence>
  <xs:element name="value" type="T_VALUE" minOccurs="0" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
```

où l'on autorise les tableaux vides *(i.e. à zéro éléments)*...

9.4.2 Structures

Les éléments de tableaux ne sont pas nommés. A contrario, une structure contient des paires nom / valeur.

L'élément *"struct"* contient une liste d'éléments *"member"*, caractérisé chacun par un élément *"name"* et un élément *"value"*.

Exemple de structure :

```
<value>
 <struct>
  <member>
   <name>Numero</name>
   <value><int>69</int></value>
  </member>
  <member>
   <name>Departement</name>
   <value><base64>Umj0bmU=</base64></value>
  </member>
  ...
 </struct>
</value>
```

L'exemple ci-dessus correspond à une structure qui comporte deux membres. L'élément *"value"* peut être de n'importe quel type, et contenir récursivement un autre élément composé *(struct ou array)*.

Remarquer comment, là encore, la valeur a été transmise en base 64, pour cause de caractères accentués. Il n'est toutefois pas possible d'utiliser la même approche pour le nom de la propriété.

> Schéma XML

Conformément à la définition ci-dessus, le schéma d'une valeur du type *"struct"* est le suivant :

```
<xs:complexType name="T_STRUCT">
 <xs:sequence>
  <xs:element name="member" maxOccurs="unbounded" type="T_STRUCT_MEMBER"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="T_STRUCT_MEMBER">
 <xs:sequence>
  <xs:element name="name" type="T_NAME"/>
  <xs:element name="value" type="T_VALUE"/>
 </xs:sequence>
</xs:complexType>
```

où l'on exige au minimum un membre dans une structure...

Devant le manque de précision des spécifications concernant les caractères autorisés pour le nom des membres de structures, on peut tenter une définition similaire à celle autorisée pour les noms de procédures :

```
<xs:simpleType name="T_NAME">
 <xs:restriction base="xs:string">
  <xs:pattern value="[A-Za-z0-9/.:_]+"/>
 </xs:restriction>
</xs:simpleType>
```

9.4.3 Schéma des valeurs XML-RPC

Le schéma de l'élément *"value"* fait apparaître le choix des divers types que cet élément est susceptible de contenir :

```
<xs:complexType name="T_VALUE" mixed="true">
 <xs:choice>
  <xs:element name="int" type="T_INT"/>
  <xs:element name="i4" type="T_INT"/>
  <xs:element name="boolean" type="T_BOOLEAN"/>
  <xs:element name="double" type="T_DOUBLE"/>
  <xs:element name="dateTime.iso8601" type="T_DATE"/>
  <xs:element name="string" type="T_STRING"/>
  <xs:element name="base64" type="T_BASE64"/>
  <xs:element name="array" type="T_ARRAY"/>
  <xs:element name="struct" type="T_STRUCT"/>
 </xs:choice>
</xs:complexType>
```

A noter que pour autoriser une valeur chaîne sans la balise *"string"* il faut que ce type soit à contenu mixte *(attribut mixed="true")* ce qui a l'inconvénient de valider des documents non valides, mais moins grave que de ne pas valider des documents conformes...

Chacun des types fait ensuite l'objet d'une définition particulière détaillée par ailleurs.

☞ Voir l'extrait de schéma complet
[http://dmolinarius.github.io/demofiles/mod-84/xrpc/values/value.xsd.html]

9.5 Format de la réponse

9.5.1 Entêtes HTTP

La réponse à une requête **XML-RPC** prend la forme d'une réponse **HTTP** classique :

```
HTTP/1.1 200 OK
Date: Mon, 17 Jan 2005 13:01:08 GMT
Server: Apache/1.3.28
X-Powered-By: PHP/4.3.3
Content-Length: 793
Connection: close
Content-Type: text/xml
```

Le code de retour doit obligatoirement être *"200 OK"*, sauf erreur bas niveau côté serveur, ce qui laisse la porte ouverte pour les codes de la famille *"500" (Internal Server Error)*. De fait, les codes de redirection *(famille 300)* sont sant objet pour **XML-RPC**, et les erreurs client *(codes 400)* sont traitées dans le corps de la réponse.

D'après les spécifications **XML-RPC** les directives *"Content-Type"* et *"Content-Length"* sont obligatoires, la valeur de *"Content-Type"* est forcément *"text/xml"*, et la valeur de *"Content-Length"* doit être correcte *(taille du corps en octets)*.

Il est possible que la réponse contienne d'autres directives, comme ici *"Connection"*, *"Server"*, ou *"X-Powered-By"*. Cette possibilité est exploitée par certaines implémentations qui sont par exemple amenées à gérer des sessions basées sur le mécanisme des *"Cookies"*.

> La directive Content-Length

Le fait que *"Content-Length"* soit obligatoire est très contraignant. En effet, ceci oblige le serveur à connaître la longueur de la réponse *(et donc à en disposer en entier)* au moment où il compose les entêtes.

D'autre part, cette directive n'est pas obligatoire en **HTTP/1.0** *(c'est la coupure de connexion qui indique la fin du message)*.

En **HTTP/1.1** la connexion n'est pas coupée, et le serveur utilise donc traditionnellement un autre mécanisme *(Content-Encoding: chunked)* qui est doublement contraire aux spécifications **XML-RPC** : d'un côté la directive *"Content-Length"* est alors absente, et d'autre part le corps de la réponse est découpé *"en tranches"* délimitées par des informations numériques sur la taille des *"tranches"*, ce qui conduit à *"polluer"* le corps :

```
HTTP/1.1 200 OK
Date: Mon, 17 Jan 2005 13:59:31 GMT
Transfer-Encoding: chunked
Content-Type: text/xml

319
<?xml version="1.0"?>
<methodResponse>
...
</methodResponse>
0
```

La solution consiste évidemment à soigneusement configurer le serveur de manière à ce qu'il génère la directive *"Content-Length"* quelle que soit la version **HTTP** de la requête, et qu'il ne se mette jamais en mode *"Content-Encoding: chunked""*.

9.5.2 Corps de la réponse

Le corps de la réponse est, comme dans le cas de la requête, composé d'un document **XML** bien formé. L'élément racine est cette fois *"methodResponse"* .

L'élément principal contient un élément *"params"*, dont le contenu autorisé est le même que pour l'élément du même nom vu dans le cadre de la requête :

```
<?xml version="1.0"?>
<methodResponse>
 <params>
  ...
 </params>
</methodResponse>
```

> Schéma XML

Le schéma XML d'une réponse standard est donc assez simple :

```
<xs:element name="methodResponse" type="T_RESPONSE"/>
<xs:complexType name="T_RESPONSE">
 <xs:sequence>
  <xs:element name="params" minOccurs="0" type="T_PARAMS"/>
 </xs:sequence>
</xs:complexType>
```

où le type *"T_PARAMS"* est celui défini pour la requête.

9.5.3 Messages d'erreur

En cas d'erreur, le serveur **XML-RPC** peut renvoyer une réponse qui comporte un élément *"fault"* en lieu et place de l'élément *"params"* :

```
<?xml version="1.0"?>
<methodResponse>
 <fault>
  ...
 </fault>
</methodResponse>
```

Le contenu de l'élément *"fault"* est forcément une seule et unique valeur *(value)*, qui correspond à une structure *(struct)* à deux membres, dont le premier s'appelle *"faultCode"* et a pour valeur un entier *(int)*, et le second s'appelle *"faultString"* et a pour valeur une chaîne de caractères.

Exemple de message d'erreur :

```
<?xml version="1.0"?>
<methodResponse>
 <fault>
  <value>
   <struct>
    <member>
     <name>faultCode</name>
     <value>
      <int>4</int>
     </value>
    </member>
    <member>
     <name>faultString</name>
     <value>
      <string>Syntax Error : found empty element value</string>
     </value>
    </member>
   </struct>
  </value>
 </fault>
</methodResponse>
```

> Schéma XML

L'élément fault est très contraint, puisque le seul degré de liberté concerne la valeur numérique du code d'erreur et la chaîne de caractères associés à l'erreur.

La rédaction d'un schéma traduisant les contraintes portant sur le contenu de la structure pose pourtant problème : il n'est pas possible de déclarer dans un même contexte deux éléments portant le même nom *(ici "member")* et dont le contenu doit être différent.

Le schéma se réduit donc à ceci :

```
<xs:complexType name="T_FAULT">
 <xs:sequence>
  <xs:element name="value" type="T_FAULT_VALUE"/>
 </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="T_FAULT_VALUE">
 <xs:sequence>
  <xs:element name="struct" type="T_FAULT_STRUCT"/>
 </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="T_FAULT_STRUCT">
 <xs:sequence>
  <xs:element name="member" type="T_STRUCT_MEMBER"/>
  <xs:element name="member" type="T_STRUCT_MEMBER"/>
 </xs:sequence>
</xs:complexType>
```

où les contraintes sur le contenu de la structure ne peuvent pas être explicitées plus avant *(tous les documents conformes seront validés, mais certains documents non conformes le seront aussi)*.

9.6 Conclusion

9.6.1 Conclusion

XML-RPC est une application **XML** conçue en 1999 pour autoriser les appels de procédures distantes via le réseau internet.

> Le pire

Elle a été conçue par une personne seule et comporte un certain nombre d'éléments dont l'implémentation aurait mérité d'être un peu mieux réfléchie. Entre autres :

- les échanges se font obligatoirement en **HTTP/1.0** *("Content-Length" obligatoire, et "Content-Encoding: chuncked" interdit,)*
- **XML-RPC** s'insère mal dans l'écosystème **XML** *(absence d'espace de noms, impossibilité de transmettre des fragments XML autrement qu'en les codant en base64)*,
- les chaînes de caractères, ont été initialement limitées aux caractères **ASCII**, avant d'autoriser *"tous les caractères"* sans plus de détails concernant les encodages autorisés ni la façon de spécifier l'encodage utilisé,
- les valeurs *(int, boolean, double, date, base64)* ne doivent pas comporter d'espaces initiaux et finaux, alors qu'il est très facile depuis n'importe quel langage de supprimer ces espaces, et ce d'autant plus que le type de chacune des valeurs est explicitement indiqué dans les messages,
- les nombres à virgule flottante sont forcément écrits en notation décimale *(les valeurs avec exposant comme 3.14E-1 sont interdites)* et certaines valeurs décrites par les spécifications **IEE 754** sont impossibles à exprimer *("NaN", "INF", "-INF")*,

Par ailleurs, de nombreux points des spécifications manquent de précision *(caractères autorisés pour les noms de procédures, directives autorisées au sein des entêtes HTTP, codages de caractères autorisés...)* ce qui laisse aux différentes implémentations le soin de résoudre les problèmes d'interopérabilité.

XML-RPC constitue à ce titre un excellent exemple de ce qu'il faut éviter de faire, et illustre parfaitement la nécessité de réfléchir en comité avant d'émettre des spécifications à vocation globale.

> Le meilleur

Toutefois, **XML-RPC** a survécu à d'autres technologies ayant tenté de résoudre le même problème *(CORBA, Java RMI, ...)* avec moins de succès car trop complexes, plus opaques, moins ouvertes et moins portables.

La force d'**XML-RPC** vient essentiellement d'**XML** qui lui garantit ouverture et portabilité, même si tous les avantages d'**XML** n'ont pas été exploités *(internationalisation, interopérabilité)*.

Le second avantage d'**XML-RPC** est lié à sa simplicité : les spécifications tiennent en une page, même si on aurait aimé parfois quelques détails supplémentaires...

> L'état de l'art

Il est aujourd'hui tout à fait possible d'utiliser **XML-RPC**, en particulier dans un contexte très orienté **XML**. Toutefois, les choix technologiques pour concevoir des appels de procédure distante, notamment dans le contexte du Web, ont actuellement tendance à privilégier le format **JSON**, plus léger qu'**XML**.