

Les *Web Services* : connecter des applications

Stéphane Bortzmeyer
AFNIC

bortzmeyer@nic.fr

\$Id: web-services.db,v 1.9 2003/11/24 09:22:53 bortzmeyer Exp \$

Copyright © 2003 AFNIC

Ce document est distribué sous les termes de la GNU Free Documentation License (<http://www.gnu.org/licenses/licenses.html#FDL>). Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

Résumé :

Faire interagir des programmes différents en réseau a toujours été une affaire complexe, notamment si on souhaite standardiser certains aspects de cette interaction (une sorte de couche à mi-chemin entre le Transport et les Applications, comme la défunte couche 5).

Les Web Services sont un ensemble de protocoles qui permettent, au moins sur le papier, de faire communiquer (avec un protocole de haut niveau, pas juste des bits) des programmes tournant sur des machines différentes et écrits dans des langages de programmation différents. Ils le font en reprenant certaines principes du Web (transport sur HTTP, formatage en XML) mais en mettant l'accent sur la communication entre applications, pas entre humains.

Ils permettent donc de connecter différents composants du système d'information (y compris entre organisations différentes).

1. Pourquoi utiliser le réseau

Pourquoi ne peut-on pas se contenter d'applications locales ? Il y a plusieurs raisons possibles.

- C'est l'autre machine qui a les données
- C'est l'autre machine qui va vite
- C'est l'autre machine qui a les bons logiciels

Ce qui est sûr, c'est qu'on n'envisage plus de tout faire sur une seule machine.

Un exemple courant est celui d'un *middleware* d'accès à une base de données : le *middleware* permet de mettre les règles du métier (*business logic*) et de pallier le manque de standardisation de SQL, ainsi que l'absence de transport standard. Le fait que le *middleware* soit client-serveur permet d'y accéder depuis d'autres systèmes et d'autres langages de programmation (contrairement aux bibliothèques, qui sont spécifiques d'un langage).

2. Avant les *Web Services*

Même si le marketing essaie de faire croire qu'avant les *Web Services* on vivait dans des cavernes, il y a longtemps que l'on fait de la programmation en réseau. Voyons quels étaient les techniques les plus utilisées.

2.1. Bricolages divers

2.1.1. Analyser HTML

L'information est souvent disponible sur une page Web. L'analyse de telles pages Web est souvent présentée comme une solution acceptable ("Mais si, nos données sont accessibles en ligne").

Dans ce cas, on doit effectuer une requête HTTP et analyser l'HTML envoyé. Il existe de très bonnes bibliothèques pour ces opérations, dans tous les langages. Leurs auteurs ont du mérite, puisque pratiquement aucun site n'envoie du HTML correct... Mais cela reste très pénible car HTML n'est pas utilisé comme langage de description de contenu mais comme langage de mise en page. L'information utile est donc noyée sous les informations de présentation.

Pire, il est fréquent que cette présentation change subitement, cassant ainsi les analyseurs.¹

En outre, il est relativement rare que l'information soit accessible directement à partir d'un URL, avec quelques paramètres (par exemple `http://www.nic.fr/cgi-bin/whois?Object=$DOMAINE`). Il faut souvent gérer une session, avec envoi de *cookies* et manipulations d'URL (ajout de *session ID* qui empêchent la réutilisation de l'URL).

L'utilisation d'HTTP n'est pas limitée à la consultation, elle permet aussi déclencher des actions par ce moyen, en appelant avec la méthode POST.

2.1.2. Analyser le non-formaté

Dans d'autres cas, on doit transmettre en non-formaté et re-analyser derrière. C'est ce que fait `whois/[11]` : il formate de l'information en texte, qu'on doit réanalyser après², alors qu'elle était structurée dans une base de données ! Certains serveurs `whois`, heureusement, formatent l'information d'une manière un peu plus lisible, avec des doublets attributs-valeurs, par exemple :

```
inetnum:      195.220.197.0 - 195.220.199.255
netname:      FR-UREC-HD
descr:        Reseau de l'Unite Reseaux du CNRS
country:      FR
admin-c:      JPG252-RIPE
tech-c:       BT261-RIPE
status:       ASSIGNED PA
mnt-by:       RENATER-MNT
changed:      rensvp@renater.fr 19990907
changed:      rensvp@renater.fr 20011031
source:       RIPE
```

1. C'est couramment une action volontaire, le *semantic firewall*, pour empêcher les récupérations automatisées.
2. Avec des bibliothèques compliquées comme `WhoisExtract` (<http://open.gandi.net/>) ou bien `Net::XWhois`.

2.2. Solutions sérieuses

Contrairement à ce que prétendent certains vendeurs de *Web Services*, il existait des solutions non bricolées.

- On faisait tout à la main (définir un protocole, écrire les clients et les serveurs)
- On avait des solutions spécifiques à un langage (RMI)
- Corba
- ONC-RPC (SunRPC)

La première approche est appréciée des techniciens, qui aiment souvent réinventer la roue. Aujourd'hui, elle ne nécessite même plus de tout refaire : BEEP simplifierait cette approche. RMI est, lui, spécifique de Java (et .COM ou .NET de Microsoft). Cela leur ôte tout intérêt puisque on utilise souvent les *Web Services* pour être indépendant du langage de programmation. Corba est trop lourd et compliqué et n'a eu aucun succès. ONC-RPC (utilisé par exemple par NFS) était techniquement très douteux (le portmapper...) mais a été largement déployé. Avec XDR, il formait une solution acceptable à l'époque où il n'y avait pas le choix. Il a popularisé le terme de RPC et, souvent, les idées de programmation distribuée.

3. Les *Web Services* arrivent

Normalisés informellement ou bien par le W3C (<http://www.w3.org/2002/ws/>), les *Web Services* représentent l'approche à la mode aujourd'hui.

Ils s'appuient sur le succès du Web :

- Disponibilité de HTTP,
- Web, donc bon,
- Et on passe les coupe-feux !

Le [13] présente un point de vue critique qui sert de base à l'IETF pour un refus général des *Web Services*³. Mais cela n'a pas empêché leur succès.

Qu'est-ce qui définit les *Web Services* ? Il n'y a pas de réponse simple, même l'utilisation de XML ne suffit pas à les caractériser. Disons que les *Web Services* comprennent :

- Un encodage (toujours XML)
- Un transport (souvent HTTP)
- Une organisation des requêtes et réponses (RPC, par exemple)

et s'appuient sur des technologies Web (serveur Apache, par exemple).

Les *Web Services* sont généralement utilisés en mode RPC. RPC veut dire *Remote Procedure Call*. C'est le modèle le plus simple en programmation distribuée. Tout est ramené à des appels de sous-programmes, avec des paramètres entrants et un résultat.

3. Les récents (RFC pas encore publié) protocoles comme EPP *Extensible Provisioning Protocol* ou bien IRIS *Internet Registry Information Service* utilisent donc XML mais sans *Web Services*.

Les *Web Services* sont un mécanisme de communication **entre applications**. Ils n'ont pas d'interface utilisateur. S'ils sont souvent désignés par un URI, ils ne sont pas accessibles à un navigateur Web classique. Un *Web Service* donné n'a d'intérêt que pour le programmeur, son existence n'est pas connue de l'utilisateur final.

4. XML-RPC, le plus simple des *Web Services*

Principe : la bibliothèque client encode les paramètres en XML et la bibliothèque serveur les décode. Le programmeur ne voit **jamais** de XML.

On ne fait que des appels de procédure : un modèle simple et bien connu.

Le transport est normalisé pour HTTP seulement, bien que des transports sur d'autres protocoles comme Jabber ou BEEP aient été mis en oeuvre.

Il existe des bibliothèques pour tous : Perl, C, Python, Ruby, Java, VisualBasic/.NET, PHP et même Emacs-Lisp.

XML-RPC, exemple Java.

```
// The server has been created above
Vector params = new Vector();
params.addElement(new Integer(5));
params.addElement(new Integer(3));
// Call the server, and get our result.
Hashtable result =
    (Hashtable) server.execute("sample.sumAndDifference", params);
// We cannot use the procedure name directly (a limit of Java), hence
// the "execute" method.
int sum = ((Integer) result.get("sum")).intValue();
int difference = ((Integer) result.get("difference")).intValue();
// Java typing makes for convoluted expressions...
```

XML-RPC, exemple Python. Le modèle de programmation de XML-RPC convient mieux aux langages dynamiques et peu typés.

```
server = xmlrpclib.Server ('http://whois.eureg.org:8080/RPC2')
# Call the server, and get our result.
result = server.sample.sumAndDifference(3, 5);
sum = result["sum"]
difference = result["difference"]
```

`sample.sumAndDifference` est une méthode. La notation pointée ne sert qu'à l'esthétique, XML-RPC ne connaît pas de hiérarchie des méthodes.

XML-RPC permet plusieurs types de paramètres :

- entiers (comme l'exemple ci-dessus), dates, booléens, chaînes de caractères
- *structs* (tableaux associatifs)
- tableaux

Dans les exemples ci-dessus, `sample.sumAndDifference` renvoyait une *struct* de deux éléments.

Les erreurs sont signalées par des exceptions.

4.1. XML-RPC, exemples de clients réels

4.1.1. Meerkat, un service d'informations en ligne

Meerkat (http://www.oreillynet.com/pub/a/rss/2000/11/14/meerkat_xmlrpc.html) est accessible en XML-RPC. Voici un exemple en PHP.

```
<?php
$server_url = '/meerkat/xml-rpc/server.php';
$msg = new xmlrpcmsg('meerkat.getCategories', array());
$client = new xmlrpc_client($server_url, "www.oreillynet.com", 80);

# Send our XML-RPC message to the server and receive a response in return
$response = $client->send($msg);
$value = $response->value();

# And convert it to a PHP data structure
$categories = xmlrpc_decode($value);

# Iterate over the results, printing each category's title
while( list($k, $v) = each( $categories ) ) {
    print $v['title'] . "<br />\n";
}

?>
```

4.1.2. Adam's Names

Une interface d'accès au registre DNS, accessible en XML-RPC (<http://www.adamsnames.tc/api/xmlrpc.html>).

Cela permet de développer un whois moderne (ici en Perl), aux résultats analysables.

```
my $rpc = Frontier::Client->new( url =>
    'http://www.adamsnames.tc/api/xmlrpc' );
my $status = $rpc->call('domquery', 'xmlrpcdemo.tc');
my $dumper = Data::Dumper->new([ $status ])->Terse(1)->Indent(1);
my $txt = $dumper->Dump;
```

4.2. XML-RPC, le serveur

Ici, un extrait d'un serveur XML-RPC dans un registre DNS. Il donne des informations sur un nom de domaine. Ce serveur utilise le *registry* de XML-RPC (pour l'introspection).

```
self.registry.add_method('registry.queryDomain',
                        self.domquery,
                        [[STRUCT, STRING, STRUCT]])
...
def domquery (self, domain, credentials):
```

```

    """Queries the registry for a domain's attributes"""
    if credentials.has_key('name'):
        raise Unauthorized
    self.cursor.execute("""
        SELECT name,
...
def call (self, methodName, params):
    """Use our registry to find and call the appropriate method."""
    try:
        return self.registry.dispatch_call(methodName, params)
    except Unauthorized:
        raise xmlrpclib.Fault(403, 'Unauthorized')

```

On enregistre la procédure `registry.queryDomain` : elle prend une chaîne et une *struct* et renvoie une *struct*.

`domquery` est une procédure normale, sans aucune connaissance de XML-RPC (par exemple, elle lève des exceptions normales).

`call` connaît le protocole et lève donc des exceptions spécifiques.

4.3. Divers

4.3.1. XML-RPC, sur le câble

Cette section n'a d'importance pratique que si vous voulez écrire une (nouvelle) bibliothèque XML-RPC ou bien si vous observez une session avec `ethereal`. Le programmeur moyen ne voit pas l'encodage en XML.

```

POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (NetBSD)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181

<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><i4>41</i4></value>
    </param>
  </params>
</methodCall>

```

Les en-têtes HTTP sont des en-têtes standard ([12]).

4.3.2. XML-RPC, limites

- En standard, chaînes en ASCII uniquement. Mais, en pratique, pas mal de mises en oeuvre de XML-RPC ont Unicode ([http://www.xmlrpc.com/discuss/msgReader\\$2129?mode=day](http://www.xmlrpc.com/discuss/msgReader$2129?mode=day)).
- Pas normalisé sous un organisme neutre (IETF, W3C, etc) ce qui est un handicap pour être utilisé comme base pour d'autres protocoles normalisés.

5. SOAP, le plus vendu

Simple Object Access Protocol est le protocole de *Web Services* le plus connu aujourd'hui. Il dispose en effet du meilleur marketing (W3C et Microsoft).

SOAP est techniquement très proche de XML-RPC. La bibliothèque client encode les paramètres en XML et la bibliothèque serveur les décode. Le programmeur ne voit **jamais** de XML.

On fait des appels de procédure, comme en XML-RPC, ou de l'asynchrone.

Il existe un grand choix de transports : en HTTP, BEEP, etc.

Le programmeur SOAP dispose d'un grand nombre de bibliothèques : Perl, C, C#, Python, Ruby, Java, VisualBasic/.NET, PHP, Ada.

SOAP, un exemple Perl.

```
# Utilise l'AUTOLOAD de Perl
use SOAP::Lite +autodispatch =>
    uri => 'http://www.soaplite.com/Temperatures',
    proxy => 'http://services.soaplite.com/temper.cgi';
print f2c(100), "\n"; # Appelle une procédure distante
```

uri identifie l'application (SOAP dit la classe et l'aiguillage vers la bonne classe se nomme *dispatching*) utilisée sur le serveur SOAP. Le même serveur peut héberger plusieurs applications. *proxy* identifie le serveur. Les deux sont des URI mais n'ont aucun rapport. Le premier est souvent un URN comme urn:GoogleSearch. Le second est plus physique : la machine nommée dans l'URL doit exister.

SOAP, un exemple Python. On utilise SOAPpy (<http://pywebsvcs.sourceforge.net/>).

```
server = SOAP.SOAPProxy('http://api.google.com/search/beta2',
                        namespace='urn:GoogleSearch')
result = server.doGoogleSearch('Zls0Q7uAt2Lrcd7BHjai...zWJj7',
                               'python wsdl', ...);
print result['estimatedTotalResultsCount']
```

La chaîne incompréhensible est la clé de la licence Google.

5.1. SOAP, détails

SOAP permet de nombreux types de paramètres :

- entiers, dates, booléens, chaînes, etc (tout ce qu'on peut décrire avec les Schémas)
- *structs* (tableaux associatifs)
- tableaux

Les erreurs sont signalées par des exceptions (*faults*).

5.2. SOAP, le serveur

```
use SOAP::Transport::HTTP;
my $daemon = SOAP::Transport::HTTP::Daemon
    -> new (LocalAddr => 'localhost', LocalPort => 8080)
```

```
-> dispatch_to('Handler');
$daemon->handle;
```

```
package Handler;
sub hi {
    return "hello, world";
}
sub bye {
    return "goodbye, cruel world";
}
```

Le serveur peut aussi être un CGI, un module mod_perl, etc.

Voici un serveur plus compliqué. Le code métier est dans un paquetage séparé. Le serveur peut recevoir des paramètres (ici \$domain).

```
# Le serveur proprement dit
use SOAP::Transport::HTTP;

my $daemon = SOAP::Transport::HTTP::Daemon
    -> new (LocalAddr => 'soap.nic.fr', LocalPort => 8080)
    -> dispatch_to(undef, Meticiel,
    undef, undef);
print "Contact to SOAP server at ", $daemon->url, "\n";
$daemon->handle;
```

```
# Le code métier
package Meticiel;

sub is_available () {
    my ($class, $domain) = shift;
    $domain = lc($domain);
    if ($domain !~ /\.fr$/) {
        return "We only register domains in \".fr\"";
    }
    if (&registered($domain)) {
        return "Domain $domain already registered";
    }
    return "Domain $domain is available. Buy it soon!";
}

sub registered () {
    ...
}
```

Un client pour ce serveur pourrait être :

```
use SOAP::Lite;

$domain = shift(@ARGV);
if (! $domain) {
    die "Usage: $0 domain-name";
}
```

```

print SOAP::Lite
  -> uri('http://soap.nic.fr/Meticiel')
  -> proxy('http://soap.nic.fr:8080/')
  -> is_available($domain)
  -> result;
print "\n";

```

Attention, la bibliothèque Perl SOAP::Lite ne lève pas d'exceptions (qui n'existent pas réellement en Perl bien qu'on puisse les simuler avec `die`). Il vaudrait donc mieux tester le code de retour avant d'appeler `result` :

```

unless ($result->fault) {
  print $result->result();
  print "\n";
} else {
  print join ', ',
    $result->faultcode,
    $result->faultstring,
    $result->faultdetail;
}

```

. Vous pouvez aussi définir le traitant `on_fault` pour appeler `die` si vous préférez les exceptions.

5.3. SOAP, sur le câble

Cela ne vous servira que si vous voulez écrire une bibliothèque.

SOAP s'appuie sur les schémas XML. SOAP permet de transmettre du XML brut (à analyser soi-même).

```

POST /StockQuote HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: 2456
SOAPAction: "http://electrocommerce.org/abc#MyMessage"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <myapp:GetLastTradePrice xmlns:myapp="Some-URI">
      <symbol>AFNIC</symbol>
    </myapp:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Les *namespaces* (ici, `myapp`) permettent de définir ses propres éléments, sans risque de collision.

5.4. SOAP, les problèmes

- Usine à gaz
- Peu interopérable

What's wrong with SOAP? SOAP est trop complexe, regardez la taille de sa spécification :

```
wc soap-spec.txt
 1519   10671   79445 soap-spec.txt
wc xmlrpc-spec.txt
   315    1657   15838 xmlrpc-spec.txt
```

5.5. SOAP, exemples de clients réels

5.5.1. Google

Google a un accès SOAP (<http://www.google.com/apis/index.html>) décrit en WSDL (inscription gratuite et obligatoire).

5.5.2. Amazon

Amazon a un accès SOAP (<http://www.amazon.com/webservices>) (inscription obligatoire mais gratuite). Cela permet d'écrire des programmes pour retrouver des informations au catalogue comme :

```
% perl amazon-by-keyword.pl ipv6
J. D. Wegner Robert Rockell Marc Blanchet Syngress Media
IP Addressing and Subnetting, Including IPv6
$41.97

Joseph Davies
Understanding IPv6
$20.99

Peter Loshin Pete Loshin
IPv6 Clearly Explained
$46.95

Regis Desmeules
Cisco Self-Study: Implementing Cisco IPv6 Networks (IPV6)
$55.00
```

...

L'API d'Amazon est assez complexe. Elle est entièrement documentée dans le SDK (*Software Development Kit*) téléchargeable sur le site d'Amazon et qui inclus un fichier WSDL. Mais le programme Perl ci-dessus ne l'a pas utilisé, il se contente du module `Net::Amazon`, disponible dans la CPAN :

```
#!/usr/bin/perl
```

```

use Net::Amazon; # In CPAN

my $keywords = (shift(@ARGV) || die "Usage: $0 keyword(s)");
my $ua = Net::Amazon->new(token => 'REGISTER_YOURSELF_DO_NOT_STEAL_MINE');
my $response = $ua->search(mode=>"books", keyword => $keywords);

if($response->is_success()) {
    foreach $property ($response->properties()) {
%book = %{$property};
foreach $author (@{$book{"authors"}}) {
    print $author, " ";
}
print "\n";
print $book{"title"}, "\n";
print $book{"OurPrice"};
print "\n\n";
    }
} else {
    print "Error: ", $response->message(), "\n";
}

```

Comme un paquetage analogue existe pour Python, ce programme pourrait s'écrire :

```

#!/usr/bin/python

import amazon, sys # http://diveintomark.org/projects/
keyword = sys.argv[1]

books = amazon.searchByKeyword(keyword)
for book in books:
    print book.Authors.Author
    print book.ProductName # Title
    print book.OurPrice
    print

```

6. UDDI, l'annuaire universel

UDDI a été normalisé par Oasis. Il permet d'enregistrer les Web Services, afin de les retrouver (on l'a décrit comme *The CPAN of Web Services*).

UDDI comprend un protocole et plusieurs registres, peut-être concurrents⁴.

Un registre UDDI peut être accédé en SOAP mais aussi en XML-RPC ou Corba.

La documentation difficile à aborder (c'est Oasis...). L'information est très structurée, avec beaucoup de niveaux (notez l'emboîtement des références dans l'exemple ci-dessous). Et la documentation n'est pas en hyper-texte :-)

UDDI, exemple.

4. Après le DNS et les certificats X509, voilà encore du travail pour les gérants de registre.

```

use UDDI::Lite +autodispatch =>
    proxy => 'http://uddi.microsoft.com/inquire';

$info = find_business(name => 'amazon')
    -> businessInfos->businessInfo->serviceInfos->serviceInfo;
print $info->name, "\n";

```

UDDI, les détails.

```

# find_* : "fuzzy" searches
# get_*   : exact searches, with the key
$mybusinessList = find_business(name => 'ama');
$mybusinessInfos = $mybusinessList->businessInfos;
@mybusinessInfo = $mybusinessInfos->businessInfo;
for $mybusinessInfo (@mybusinessInfo) {

    print $mybusinessInfo->name, "\n";
    print $mybusinessInfo->businessKey, "\n\n";

    $myserviceInfos = $mybusinessInfo->serviceInfos;
    @myserviceInfo = $myserviceInfos->serviceInfo;

    for $myserviceInfo (@myserviceInfo) {
print "    ", $myserviceInfo->name, "\n";
print "    ", $myserviceInfo->serviceKey, "\n";
@myserviceDetails = get_serviceDetail
    (serviceKey => $myserviceInfo->serviceKey);
for $myserviceDetail (@myserviceDetails) {
    print "        ", $myserviceDetail->name, "\n";
    print "        ", $myserviceDetail->description, "\n";
    $mybindingTemplate = $myserviceDetail->bindingTemplates->bindingTemplate; # Actually, several
    print "            ", $mybindingTemplate->description, "\n";
    print "            ", $mybindingTemplate->accessPoint, "\n";
    }
print "\n";
    }

    print "\n\n";
}

```

7. WSDL, méta-informations

WSDL est un langage du W3C pour décrire les API (*Application Programming Interface*) des *Web Services* (surtout pour SOAP). Il est décrit en XML.

WSDL est complexe car il permet de décrire plusieurs modèles d'interactions, pas seulement le classique RPC. Comme il s'appuie sur les schémas XML (notamment pour les types de données), il faut connaître les schémas d'abord.

WSDL peut servir de documentation formelle à vos *Web Services*, documentation que vous pouvez ensuite présenter joliment grâce à des feuilles de style (<http://www.capescience.com/articles/simplifiedWSDL/>) mais on peut aussi écrire des clients qui analysent le WSDL et trouvent ainsi "tout seul" la marche à suivre pour utiliser le *Web Service*.

WSDL, un extrait.

```

<xsd:complexType name="ResultElement">
  <xsd:all>
    <xsd:element name="summary" type="xsd:string"/>
    <xsd:element name="URL" type="xsd:string"/>
    <xsd:element name="snippet" type="xsd:string"/>
    <xsd:element name="title" type="xsd:string"/>
  ...
</xsd:complexType>
...
<message name="doGoogleSearch">
  <part name="key" type="xsd:string"/>
  <part name="q" type="xsd:string"/>
  ...
</message>

<message name="doGoogleSearchResponse">
  <part name="return" type="typens:GoogleSearchResult"/>
</message>

```

WSDL, exemple d'utilisation.

```

my $google = SOAP::Lite->service('http://api.google.com/GoogleSearch.wsdl');
my $result = $google->doGoogleSearch(
  $key, $query, 0, 10, 'false', "", 'false', "", 'latin1', 'latin1');

```

Ici, le client Perl a trouvé le type des paramètres uniquement en utilisant le fichier `GoogleSearch.wsdl`.

8. Utiliser un *Web Service*, quelques conseils

Pour mettre en harmonie tous les concepts vus jusqu'à présent, essayons l'utilisation d'un *Web Service* réel. Beaucoup de *feeds* de nouvelles sont disponibles sur Internet, utilisant en général l'une des variantes du langage RSS (<http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html>) (Meerkat en fait partie). La difficulté est de trouver le bon *feed*. Certains offrent des services de recherche de *feed* et Syndic8 (<http://www.syndic8.com/>), que nous utilisons ici, offre une interface *Web Services*, ici XML-RPC.

Avant toute utilisation d'un *Web Services* il faut évidemment avoir appris à écrire un client, même trivial, dans son langage de programmation favori, et il faut avoir donc choisi une bibliothèque (par exemple, en Perl, il existe trois bibliothèques pour faire du XML-RPC).

Ensuite, la première étape est de lire la spécification du service. (Cette étape peut être partiellement sautée si le service est décrit en WSDL.) Sinon, la spécification est en général une page Web comme celle de Syndic8 (<http://www.syndic8.com/services.php>). Pour notre service convoité, on apprend que Syndic8 permet de trouver les *feeds* en indiquant un motif (comme "Internet" ou bien "France"), c'est le service `syndic8.FindFeeds` et donne des informations détaillées sur les *feeds* si on lui fournit l'index (*ID*) de ceux-ci, c'est le service `syndic8.GetFeedInfo`.

On trouve également dans la documentation l'URL (*end point*) du service, ici <http://www.syndic8.com/xmlrpc.php>.

On peut alors écrire un programme, ici en Python :

```

#!/usr/bin/python

import xmlrpclib, socket

```

```

server = "http://www.syndic8.com/xmlrpc.php"
pattern = "france"
try:
    handle = xmlrpclib.Server (server)
    feeds = handle.syndic8.FindFeeds(pattern)
    for feed in feeds:
        info = handle.syndic8.GetFeedInfo (feed)
        print "Feed %s:" % feed
        print info
except socket.error, message:
    print "Cannot contact the server: " + str(message)
except xmlrpclib.ProtocolError, message:
    print "The server refused to reply: " + str(message)
except xmlrpclib.Fault, message:
    print "Error/bug inside the server: " + str(message)

```

Ce premier programme a plusieurs limites :

1. Il imprime tout le résultat de `syndic8.GetFeedInfo`, un *struct*, alors que tous les champs ne sont pas utiles. On note que les champs de ce tableau associatif ne sont pas documentés, hélas, mais leur nom est en général clair.
2. Il imprime tous les *feeds* correspondant à notre motif "France" alors que, dans ce genre de services, la majorité des *feeds* indiqués sont hors service. Syndic8 les détecte et met un statut utile dans la *struct*.
3. Les champs de texte sont en Unicode (<http://www.unicode.org/>) ce qui est habituel dans le monde XML mais Python ne traite pas l'Unicode par défaut.

Voici une nouvelle version, qui n'imprime les *feeds* que si leur statut est positif et qu'ils portent suffisamment d'articles. D'autre part, on n'imprime plus tous les champs mais seulement la description et l'URL où on pourra récupérer l'élément RSS. Enfin, on gère correctement l'Unicode.

```

#!/usr/bin/python

import xmlrpclib, socket

server = "http://www.syndic8.com/xmlrpc.php"
pattern = "france"
try:
    handle = xmlrpclib.Server (server, verbose=0)
    feeds = handle.syndic8.FindFeeds(pattern)
    for feed in feeds:
        info = handle.syndic8.GetFeedInfo (feed)
        if info["status"] == "Syndicated" and \
            float(info["headlines_per_day"]) > 5: # Only keep those that are active
            print "Feed %s:" % feed
            desc = unicode (info["description"]) # Turn it into an
                                                    # Unicode string.
            print desc.encode ("latin-1"),      # Print what my
                                                    # terminal supports, here Latin-1.

            print " (" + info["dataurl"] + ")"
except socket.error, message:
    print "Cannot contact the server: " + str(message)
except xmlrpclib.ProtocolError, message:

```

```
print "The server refused to reply: " + str(message)
except xmlrpclib.Fault, message:
    print "Error/bug inside the server: " + str(message)
```

Ce programme peut alors afficher :

```
Feed 369:
France News (http://p.moreover.com/cgi-local/page?index_france+rss)
```

9. Déployer un *Web Service*, quelques conseils

Avant de déployer un *Web Service* quelques points sont à considérer. La plupart relèvent du bon sens mais il est prudent des les mentionner explicitement.

9.1. Spécifier le service

Cela implique une réflexion sur l'API qui sera présentée aux utilisateurs. Comme changer l'API nécessiterait un changement de tous les clients, il faut tenter de réussir l'API du premier coup, probablement en testant ses versions bêta avec de vrais utilisateurs.

Il faudra aussi choisir un protocole, XML-RPC ou bien SOAP.

9.2. Sécurité

Un *Web Service* n'est pas forcément accessible depuis l'extérieur de votre organisation. Mais, s'il l'est, la sécurité est essentielle. S'il fonctionne qu'en lecture seule, n'agissant en rien sur vos données, il pourra être anonyme. S'il modifie vos données, il faudra un mécanisme d'authentification, soit bâti dans votre application, soit récupéré dans l'environnement extérieur (Apache, par exemple, si vous utilisez SOAP en CGI ou en module Apache).

On pourra consulter l'article sur SOAP dans Cryptogram (<http://www.schneier.com/crypto-gram-0006.html#SOAP>).

9.3. Composants métier

Naturellement, le but du *Web Service* est de donner accès à des applications spécifiques de votre métier. Ces applications ont souvent été écrites avant le *Web Service*. Si elles sont sous forme de bibliothèques bien écrites (API claire, pas d'effets de bord), elles pourront être appelées directement par le *Web Service*.

10. Vers une généralisation des *Web Services* ?

Web Service est désormais un terme à la mode : il apparaît dans tous les nouveaux projets informatiques, des livres lui sont consacrés⁵. Comme tous les termes à la mode, l'abondance de références n'a d'égale que le petit nombre de

5. 739 à Amazon aujourd'hui.

projets effectivement déployés. Encore que ce nombre n'est pas facile à mesurer, une grande partie des déploiements étant uniquement faits en interne.

Y aura t-il un jour déploiement massif des *Web Services* ? La réponse n'est pas simple car elle dépend de deux choses :

1. La migration des services réseaux depuis Corba, ONC-RPC ou depuis les protocoles privés, vers les techniques XML. C'est une décision surtout technique et les *Web Services* ont de bons arguments ici.
2. L'exposition par les organisations de leur Système d'Information interne, sous forme de *Web Services* accessibles depuis l'extérieur.

Et, là, c'est beaucoup plus incertain : cela nécessiterait de surmonter de nombreux blocages non techniques, notamment la culture de la rétention d'information. Les techniques existantes, peu pratiques, ont justement cet avantage pour beaucoup de décideurs : elles rendent l'accès à l'information plus difficile. [4] expose une partie des problèmes qui se posent.

Bibliographie

[1] beepcore.org, *BEEP Home Page*, ?.

<http://www.beepcore.org/>

[2] Marshall T. Rose, *BEEP: The Definitive Guide*, 2002, O'Reilly.

[3] Object Management Group, *Corba Home Page*, ?.

<http://www.corba.org/>

[4] Christopher Koch, *The battle for Web Services*, 2003.

<http://www.cio.com/archive/100103/standards.html>

[5] UserLand Software, *XML-RPC Home Page*, 2003.

<http://www.xml-rpc.com/>

[6] Eric Kidd, *XML-RPC introspection protocol*, 2001.

<http://xmlrpc-c.sourceforge.net/xmlrpc-howto/xmlrpc-howto-api-introspection.html>

[7] Simon St. Laurent, Joe Johnston, Edd Dumbill, *Programming Web Services with XML-RPC*, 2001, O'Reilly.

[8] Userland Software, *SOAP Home Page*, 2003.

<http://www.soapware.org/>

[9] Paul Kulchenko, *SOAP::Lite for Perl*, 2003.

<http://www.soaplite.com/>

[10] James Snell, Doug Tidwell, Pavel Kulchenko, *Programming Web Services with SOAP*, 2001, O'Reilly.

[11] K. Harrenstien, M. Stahl, E. Feinler, *RFC 0954: NICNAME/WHOIS*, 1985.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1*, 1999.

[13] K. Moore, *RFC 3205: On the use of HTTP as a Substrate*, 2002.