RFC 5734 : Extensible Provisioning Protocol (EPP) Transport over TCP

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 22 octobre 2009. Dernière mise à jour le 30 octobre 2009

Date de publication du RFC : Août 2009

http://www.bortzmeyer.org/5734.html

Ce court RFC spécifie comment utiliser le protocole d'avitaillement EPP au dessus d'une simple connexion TCP.

EPP, décrit dans le RFC 5730 [1] est à sa base uniquement un format XML pour les requêtes d'avitaillement (création, mise à jour et destruction d'objets) et leurs réponses. Ces éléments XML peuvent être transmis de différence façon (au dessus de HTTP, de BEEP, par courrier électronique, etc), et notre RFC normalise la plus commune aujourd'hui, une simple connexion TCP. Il remplace le RFC 4934, avec uniquement des modifications de détail, portant notamment sur l'utilisation de TLS (section 9).

Le RFC est court car il n'y a pas grand'chose à dire, juste l'utilisation des primitives de TCP (ouverture et fermeture de connexion, section 2 du RFC), l'ordre des messages (section 3), le port utilisé (700, 3121 ayant été abandonné, section 7) et le fait que chaque élément EPP soit précédé d'un entier qui indique sa taille (section 4). Sans cet entier (qui joue le même rôle que l'en-tête `Content-Length` de HTTP), il faudrait, avec la plupart des implémentations, lire les données octet par octet (sans compter que la plupart des analyseurs XML ne savent pas analyser de manière incrémentale, il leur faut tout l'élément). En outre, sa présence permet de s'assurer que toutes les données ont été reçues (voir l'excellent article "*The ultimate SO_LINGER page, or : why is my tcp not reliable*" <http://blog.netherlabs.nl/articles/2009/01/18/the-ultimate-so_linger-page-or-why-is-my-tcp-not-reliable>).

L'entier en question est fixé à 32 bits. Si on programme un client EPP en Python, l'utilisation brutale du module struct <http://docs.python.org/lib/module-struct.html> ne suffit pas forcément. En effet :

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc5730.txt

```
struct.pack("I", length)
```

force un entier (`int`) mais pas forcément un entier de 32 bits. Pour forcer la taille, il faut utiliser également, comme précisé dans la documentation, les opérateurs ¡ et ¿, qui servent aussi à forcer la boutianité (merci à Kim-Minh Kaplan pour son aide sur ce point). Voici une démonstration (un "l" standard fait 4 octets alors que le type long de C peut faire 4 ou 8 octets) :

```
# Machine 32 bits :

Python 2.4.4 (#2, Apr  5 2007, 20:11:18)
[GCC 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import struct
>>> print struct.calcsize("l")
4
>>> print struct.calcsize(">l")
4
```

```
# Machine 64 bits :

Python 2.4.5 (#2, Mar 11 2008, 23:38:15)
[GCC 4.2.3 (Debian 4.2.3-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import struct
>>> print struct.calcsize("l")
8
>>> print struct.calcsize(">l")
4
```

Si on a quand même un doute, on peut tester la taille obtenue mais ce code est probablement inutile (merci à David Douard pour son aide ici) :

```
# Get the size of C integers. We need 32 bits unsigned.
format_32 = ">I"
if struct.calcsize(format_32) < 4:
    format_32 = ">L"
    if struct.calcsize(format_32) != 4:
        raise Exception("Cannot find a 32 bits integer")
elif struct.calcsize(format_32) > 4:
    format_32 = ">H"
    if struct.calcsize(format_32) != 4:
        raise Exception("Cannot find a 32 bits integer")
else:
    pass
...
def int_from_net(data):
    return struct.unpack(format_32, data)[0]

def int_to_net(value):
    return struct.pack(format_32, value)
```

L'algorithme complet d'envoi est :

```
epp_as_string = ElementTree.tostring(epp, encoding="UTF-8")
# +4 for the length field itself (section 4 mandates that)
# +2 for the CRLF at the end
length = int_to_net(len(epp_as_string) + 4 + 2)
self._socket.send(length)
self._socket.send(epp_as_string + "\r\n")
```

et la lecture :

```
data = self._socket.recv(4) # RFC 5734, section 4, the length
                            # field is 4 bytes long
length = int_from_net(data)
data = self._socket.recv(length-4)
epp = ElementTree.fromstring(data)
if epp.tag != "{%s}epp" % EPP.NS:
     raise EPP_Exception("Not an EPP instance: %s" % epp.tag)
xml = epp[0]
```

Le code Python complet (qui ne met en œuvre qu'une petite partie de EPP, le but était juste de tester ce RFC 5734), utilisant la bibliothèque ElementTree <http://effbot.org/zone/element-index.htm>, est disponible en ligne (en ligne sur http://www.bortzmeyer.org/files/epp-python.tar.gz). Le code ci-dessus comporte une grosse faiblesse (mais classique) : rien ne garantit que `recv(n)` retournera **autant** d'octets que réclamé. Il en renverra **au plus** n mais peut-être moins. Pour de la programmation sérieuse, il faut donc le réécrire avec une fonction du genre :

```
   def safe_recv(s, n):
       data = ''
       while (n > 0):
           tmp = s.recv(n)
           data += tmp
           n -= len(tmp)
       return data
```

(Merci à Kim-Minh Kaplan pour son aide sur ce point.)

Pour Perl, ce code ressemblerait (merci à Vincent Levigneron), pour envoyer un élément EPP stocké dans la variable `$out`, à :

```
print pack('N', length($out) + 4).$out;
```

et pour lire un élément EPP :

```
my $length = unpack "N", $buf;
...
$rc = read STDIN, $buf, $length;
```

Puisque le format `N` désigne un entier non signé gros boutien (cf. <`http://perldoc.perl.org/functions/pack.html`>).

À noter que cette lecture du champ longueur présente un risque de sécurité : si le serveur `malloque` aveuglément la taille indiquée dans ce champ, comme il fait quatre octets, le serveur naïf risque de consommer quatre giga-octets de mémoire.

La section 8, consacrée à la sécurité, et surtout la nouvelle section 9, consacrée à TLS, détaillent le processus d'authentification du client et du serveur. L'utilisation de TLS (RFC 5246) est obligatoire, ne serait-ce que pour protéger les mots de passe qui circulent en clair dans les éléments EPP. TLS permet également au client d'authentifier le serveur et au serveur d'authentifier le client, par la présentation de certificats X.509. Leur présentation et leur usage sont désormais obligatoires dans EPP. Notons que, comme toujours avec X.509, la difficulté est de déterminer ce qu'on vérifie. La section 9 détaille le cas où on vérifie un nom et celui où on vérifie une adresse IP. Il serait intéressant de savoir quels registres et quels bureaux d'enregistrement effectuent réellement ces validations...

La liste des changements par rapport au RFC 4934 se trouve dans l'annexe A. Le principal changement consiste en une meilleure spécification des règles de vérification du certificat X.509 lorsque TLS est utilisé (cf. la nouvelle section 9).