

WebDAV, du HTTP AU PARTAGE DE FICHIERS

PREDRAG.VICEIC@epfl.ch, DOMAINE IT



Sinon tous, du moins la plus grande partie des lecteurs de ce journal connaissaient le protocole HTTP. À l'origine du Web et utilisé bien avant sa standardisation en 1996 (RFC 1945, facile à retenir), ce protocole basique n'a dans l'opinion de beaucoup que très peu évolué depuis sa conception. Utilisé principalement comme transport du langage HTML, c'est essentiellement à l'évolution de ce dernier et son interaction avec Javascript qu'on attribue les avancées des technologies du Web les plus récentes. Durant tout ce temps, le protocole HTTP est resté le serviteur humble se contentant de transporter du contenu à l'aide de ses 3 méthodes: HEAD, GET et POST.

Pourtant, tapis dans l'ombre et invisible, le HTTP gronde. Une révolution silencieuse est à l'œuvre depuis quelques années. Cette révolution a commencé en 1999 avec la définition de quelques extensions obscures au protocole HTTP et aujourd'hui largement méconnues du grand public, aussi technique fût-il. Le HTTP inoffensif auquel les routeurs font confiance, le HTTP un peu simplet que les firewalls laissent passer, le HTTP risible qu'on crypte si l'on veut et qu'on authentifie si on a le temps, ce HTTP-là a entrepris sa mue. Et dans sa marche, il se pourrait bien qu'il balaie tout devant lui. Car pourquoi avoir tous ces protocoles, si on peut garder que le plus beau ?

Mais avant cela, revenons quelques années en arrière. J'essaierais tout d'abord, à travers les exemples, d'illustrer le fonctionnement du protocole HTTP pour ensuite parler des extensions de celui-ci. Je montrerai comment ces extensions fournissent les outils simples, mais performants pour la gestion des fichiers, des versions, les droits d'accès, recherche, etc. Vous pourrez suivre la plupart de ces exemples à l'aide de l'application `telnet` ou `netcat`, directement depuis votre poste. Les utilisateurs windows devront exécuter le programme `cmd.exe` (**Démarrer->Exécuter** ensuite taper `cmd.exe` puis **Return**), pour les autres, ils lanceront le terminal de leur choix. Si vous ne désirez pas essayer par vous-mêmes tant pis, merci de me faire confiance.

GET

Le protocole HTTP fonctionne selon le schéma requête->traitement->réponse. Voici un exemple de la requête:

```
GET http://www.epfl.ch/ HTTP/1.0↵
```

On y voit la méthode (GET), l'URI du contenu à récupérer (/) ainsi que la version du protocole. Quand le serveur reçoit cette requête, il la traite et retourne la réponse:

```
HTTP/1.1 200 OK
Date: Tue, 03 Jun 2008 09:16:08 GMT
[...]
Content-Type: text/html
Content-Language: fr-ch
```

```
<!DOCTYPE HTML PUB [...]
[...]↵
```

La réponse commence par un statut (200 OK), ensuite viennent quelques en-têtes destinés au client Web (Firefox, IE) et finalement on reçoit le contenu se trouvant à l'URI mentionné. Vous pouvez essayer ceci par vous-mêmes en lançant, depuis votre terminal:

```
telnet www.epfl.ch 80↵
GET http://www.epfl.ch HTTP/1.0↵
↵
```

Au retour, vous aurez la page Web de l'école, en langage HTML.

La requête peut également contenir les en-têtes qui seront rajoutés à la ligne après la méthode (GET) et ses paramètres:

```
GET http://www.epfl.ch HTTP/1.0↵
Accept-Language: en↵
```

Les en-têtes nous donnent les moyens d'envoyer un peu plus d'informations au serveur, dans ce cas précis c'est la langue désirée du document. Dans la réponse de notre exemple, vous pourrez discerner, à travers les brumes de HTML, la version anglaise de la page principale du site de l'école.

HEAD

La méthode HEAD est encore plus simple. En l'exécutant comme suit:

```
telnet www.epfl.ch 80↵
HEAD http://www.epfl.ch HTTP/1.0↵
↵
```

Nous obtenons:

```
HTTP/1.1 200 OK
Date: Tue, 03 Jun 2008 11:05:43 GMT
Server: Apache/2.0.52 (Red Hat)
Content-Location: index.fr.html
Vary: negotiate,accept-language
TCN: choice
Last-Modified: Mon, 02 Jun 2008 10:48:39 GMT
ETag: "22e51-354e-bcfecfc0"
Accept-Ranges: bytes
Content-Length: 13646
Connection: close
Content-Type: text/html
Content-Language: fr-ch
```

Eh oui, ce sont uniquement les en-têtes de la réponse, sans le contenu, le client Web pourra, en y analysant les valeurs ETag ou *Last-Modified* savoir s'il doit télécharger la page ou s'il peut garder la version récupérée lors d'une requête précédente. La méthode HEAD est la plus simple de toutes et nous ne parlerons probablement plus de celle-ci.

POST

La méthode suivante, la méthode POST, nous permet d'envoyer les formulaires:

```
telnet search.epfl.ch 80↵
POST http://search.epfl.ch/directory.do
HTTP/1.0↵
Content-Length:11↵
Content-type: application/x-www-form-
urlencoded↵
↵
name=viceic↵
```

En retour, nous obtenons le résultat fourni habituellement par l'interface disponible sur *search.epfl.ch*. Dans ce cas précis, il s'agit de la page avec les résultats de recherche d'une personne dans l'annuaire. Les champs du formulaire – le mot clé dans notre cas – se trouvent dans le corps du message. Le corps du message commence après une ligne vide suivant les en-têtes. L'en-tête *Content-Length* spécifie le nombre de caractères (octets) qui suivent le retour à la ligne. Ceci permet d'indiquer au serveur la fin de la transmission des données.

Malheureusement pour la méthode POST, nous pouvons obtenir le même résultat avec la méthode GET:

```
telnet search.epfl.ch 80↵
GET http://search.epfl.ch/directory.
do?name=viceic HTTP/1.0↵
```

Les manuels différencient le POST et le GET dans le cadre d'envoi d'un formulaire par le fait que GET doit être utilisé lors d'une opération idempotente. Il s'agit d'une opération qui peut être exécutée plusieurs fois sans conséquence, car ne modifiant pas la base de données – une recherche par exemple.

Le POST doit par contre être utilisé quand l'opération modifie les choses plus loin - la commande en ligne d'un article par exemple - et dont la répétition, par exemple un rechargement par le navigateur, pourrait avoir les conséquences indésirables: la répétition du paiement par carte de crédit.

D'un point de vue un peu plus concret, la première différence entre les deux méthodes est que dans la méthode POST, les champs du formulaire doivent être dans le corps de la requête -commençant après une ligne vide-, alors que lors de l'utilisation de la méthode GET, ces paramètres peuvent être appendus à l'URI: ils viennent dans ce cas après le signe `?`. Ainsi, pour envoyer les données du client vers le serveur on devrait utiliser la méthode POST à la place de la méthode GET car le corps de la requête peut contenir un nombre illimité de données et ainsi permettre la construction de très gros formulaires. Quant à la taille de l'URI, elle est parfois limitée par les clients Web (IE pour ne pas le nommer: support.microsoft.com/kb/q208427/). La méthode POST n'ayant ainsi pas de limites quant à la quantité de données envoyées, on l'utilisera également pour télécharger les fichiers vers le serveur - en encodant leur contenu dans le corps du message.

La deuxième différence entre ces deux méthodes – et pour les coupeurs de cheveux en quatre d'entre nous, fondamentale – est que si l'utilisateur demande au client Web de renvoyer une requête POST, en rechargeant la page avec les résultats d'un formulaire par exemple, le client doit l'avertir qu'il risque de renvoyer le formulaire une deuxième

fois – et par là peut-être passer une commande deux fois. C'est le moyen dont votre client Web dispose afin de vous empêcher de reexécuter involontairement une opération non idempotente. Il est toutefois très rare qu'on tombe sur un site de e-commerce qui n'aurait pas ses propres mécanismes pour empêcher ce genre de mésaventures.

Voici donc un bref parcours du protocole HTTP/1.0. Basique non? Ce qui est important de retenir c'est que avec ces trois commandes: HEAD, GET et POST, la possibilité de préciser le contexte de la requête ou de la réponse via les en-têtes et la possibilité d'envoi d'un flux de données dans le corps du message de la requête ou de la réponse, nous arrivons à faire communiquer des millions de machines à travers le monde. La simplicité du protocole est justement sa force. La difficulté se trouve évidemment au niveau des interpréteurs du langage HTML, mais HTTP, lui a su rester simple. Je ne parle à dessein pas du HTTP/1.1 car il ne rajoute que peu de fonctionnalités importantes.

WEBDAV

Le protocole WebDAV (*Distributed Authoring and Versioning*) existe, du moins sur le papier, depuis 1999 (RFC 2518). Comme son nom l'indique, l'objectif du protocole est de permettre l'édition à plusieurs de documents Web. Concrètement, le protocole WebDAV fournit les méthodes de lecture/écriture des métadonnées sur les ressources Web tout en permettant de gérer ces ressources sous forme d'une arborescence de dossiers/fichiers. La simultanéité des accès en écriture dans les fichiers (édition à plusieurs) est gérée par les mécanismes de verrous. Les métadonnées, qui ne sont rien de plus qu'une sorte d'étiquettes virtuelles, peuvent être rajoutées à tout type de fichiers ou de dossiers. Ces métadonnées peuvent servir pour indiquer la date de la dernière édition, l'auteur, les contributeurs, le statut de publication du document, etc.

WebDAV (tools.ietf.org/html/rfc2518), pour ne rien vous cacher, suit bien la philosophie décennale de l'un de ses créateurs: *Embrace and Extend*. Il est mis en œuvre en rajoutant les nouvelles méthodes aux HEAD, GET et POST existantes. Le corps du message, vu plus haut, a été adapté à ces nouvelles méthodes, en remplaçant le texte libre par du XML, plus facilement structurable. Enfin, quelques nouveaux en-têtes ont été définis et voilà, le tour est joué. En prétendant améliorer le protocole HTTP, on obtient une version dopée de ce dernier, mais uniquement compréhensible par les serveurs IIS de Microsoft...

Non, non, je rigole... À vrai dire, Microsoft a un peu abandonné le WebDAV à lui même - à nous en fait - ce qui, tout en retardant son implémentation efficace, a permis de lui trouver une autre utilisation: gestionnaire de fichiers en ligne. La partie *en ligne* me semble claire. Je ne prends pas beaucoup de risques en affirmant que le HTTP est le protocole le mieux supporté actuellement sur la toile. WebDAV n'est que le HTTP avec un peu de formalisme structurel en plus, ce qui fait qu'il est vu par les acteurs présents lors d'acheminement de données via internet - providers, cablo-opérateurs, NSA etc. – comme du HTTP. Si votre ordinateur peut accéder au Web, il peut accéder au WebDAV (je l'aime bien celle-là ..).

La partie *Gestionnaire de fichiers* deviendra plus claire lorsque j'aurais décrit ces nouvelles méthodes issues de la spécification WebDAV.

PROPFIND

Rentrons dans le vif du sujet en parlant tout d'abord de la méthode PROPFIND, la substantifique moëlle du protocole WebDAV:

Requête

```
PROPFIND http://documents.epfl.ch/users/p/pv/
pviceic/www/test_webdav HTTP/1.1
Host: documents.epfl.ch
Depth:1
Content-Length:114

<?xml version="1.0" encoding="utf-8"?>
<D:propfind xmlns:D="DAV:"><D:prop><D:displayname/></D:prop></D:propfind>
```

Réponse

```
HTTP/1.1 207 Multi-Status
[.]
<D:response>
<D:href>http://documents.epfl.ch/users/p/pv/
pviceic/www/test_webdav/dossierB/</D:href>
<D:propstat>
<D:prop>
<D:displayname><![CDATA[dossierB]]></
D:displayname>
</D:prop>
<D:status>HTTP/1.1 200 OK</D:status>
</D:propstat>
</D:response>
[.]
<D:response>
<D:href>http://documents.epfl.ch/users/p/pv/
pviceic/www/test_webdav/dossierA/</D:href>
[.]
<D:displayname><![CDATA[dossierA]]></
D:displayname>
[.]
</D:response>
<D:response>
<D:href>http://documents.epfl.ch/users/p/pv/
pviceic/www/test_webdav/dossierC/</D:href>
[.]
<D:displayname><![CDATA[dossierC]]></
D:displayname>
[.]
</D:response>
[.]
```

L'exemple ci-dessus est parfaitement exécutable depuis votre poste (du moins jusqu'à ce que je n'ai pas supprimé le répertoire test_webdav). Le serveur que j'utilise comme exemple est documents.epfl.ch, le serveur WebDAV du projet my.epfl. L'exemple nous montre que PROPFIND, comme son nom l'indique - PROProperty FIND - nous sert à récupérer les propriétés (displayname) des ressources WebDAV (/users/p/pv/pviceic/www/test_webdav). L'en-tête **Depth:1** spécifie qu'on veut la ressource et ses sous-ressources (les sous-dossiers) jusqu'à la profondeur 1, donc les descendants immédiats. Nous voyons donc que la ressource **test_webdav** contient 3 sous-ressources dont le nom est dossierA, dossierB et dossierC.

En guise de parenthèse, vous remarquerez que contrairement aux exemples précédents, la connexion avec le serveur n'est pas fermée immédiatement – application telnet ne quitte pas (si vous aviez exécuté les exemples, vous l'auriez remarqué ☺). Cela nous permet de continuer à envoyer les requêtes sans ouvrir de nouvelles connexions. Ce comportement est dû aux modifications apportées par la version 1.1 du protocole HTTP. HTTP/1.1, contrairement à son prédécesseur, ne ferme pas immédiatement la connexion afin de minimiser les délais de latence présents lors d'établissement de celle-ci. Si vous voulez que le serveur ferme la connexion dès la réception de la réponse, il vous faut rajouter l'en-tête Connection: close.

Dans l'exemple qui suit, nous demandons les propriétés *resourcetype* et *creationdate*:

Requête

```
PROPFIND http://documents.epfl.ch/users/p/pv/
pviceic/www/test_webdav HTTP/1.1
Host: documents.epfl.ch
Connection: close
Depth:1
Content-Length:132

<?xml version="1.0" encoding="utf-8"?>
<D:propfind xmlns:D="DAV:"><D:prop><D:creationdate/><D:resourcetype/></D:prop></D:propfind>
```

Réponse

```
[.]
<D:response>
<D:href>http://documents.epfl.ch/users/p/pv/
pviceic/www/test_webdav/dossierB/</D:href>
[.]
<D:creationdate>2008-06-03T10:41:30Z</
D:creationdate>
<D:resourcetype><D:collection/></
D:resourcetype>
[.]
</D:response>
<D:response>
<D:href>http://documents.epfl.ch/users/p/pv/
pviceic/www/test_webdav/</D:href>
[.]
<D:creationdate>2008-06-03T10:41:12Z</
D:creationdate>
<D:resourcetype><D:collection/></
D:resourcetype>
[.]
</D:response>
<D:response>
<D:href>http://documents.epfl.ch/users/p/pv/
pviceic/www/test_webdav/dossierA/</D:href>
[.]
<D:creationdate>2008-06-03T10:41:18Z</
D:creationdate>
<D:resourcetype><D:collection/></
D:resourcetype>
[.]
</D:response>
<D:response>
<D:href>http://documents.epfl.ch/users/p/pv/
pviceic/www/test_webdav/dossierC/</D:href>
[.]
<D:creationdate>2008-06-03T10:41:34Z</
D:creationdate>
<D:resourcetype><D:collection/></
D:resourcetype>
[.]
</D:response>
[.]
```

La réponse nous indique que les trois sous-ressources (dossier A, B et C) ainsi que la ressource test_webdav sont les collections - un nom pompeux pour dénoter les dossiers. Elle nous indique également la date de leur création. Si nous voulons récupérer toutes les propriétés disponibles pour la ressource, il suffit d'exécuter:

```
PROPFIND /users/p/pv/pviceic/www/test_webdav
HTTP/1.1↵
Host: documents.epfl.ch↵
Depth:1↵
Content-Length:93↵
↵
<?xml version="1.0" encoding="utf-8" ?>↵
<D:propfind xmlns:D="DAV:"><D:allprop/></
D:propfind>↵
```

La méthode PROPFIND permet donc de récupérer les *étiquettes* présentes sur les ressources tout en permettant l'affichage de l'arborescence de ses ressources, à l'aide de l'en-tête *Depth* (profondeur). La profondeur peut être de **0** (la ressource uniquement), **1** (les sous-dossiers immédiats) et **Infinity** pour récupérer toute l'arborescence.

PROPPATCH

PROPPATCH (PROPerTy PATCH) est la méthode qui permet de placer les étiquettes sur les ressources:

Requête

```
PROPPATCH /users/p/pv/pviceic/www/test_
webdav/dossierA HTTP/1.1↵
Host: documents.epfl.ch↵
Connection: close↵
Content-Length:176↵
↵
<?xml version="1.0" encoding="utf-8" ?>↵
<D:propertyupdate xmlns:D="DAV:"><D:set
><D:prop><D:couleur-de-cheveux>blonds</
D:couleur-de-cheveux></D:prop></D:set></
D:propertyupdate>↵
```

Réponse

```
[..]
<D:response>
<D:href>http://documents.epfl.ch/users/p/
pv/pviceic/www/test_webdav/dossierA/</
D:href><D:propstat>
<D:prop><D:couleur-de-cheveux
XSI:type="XS:string"/></D:prop>
<D:status>HTTP/1.1 200 OK</D:status>
[..]
</D:response>
[..]
```

Si on re-essaye PROPFIND vu plus haut, on obtient la réponse:

```
[..]
<D:response xmlns:ns-1="http://www.xythos.
com/namespaces/StorageServer">
<D:href>http://documents.epfl.ch/users/p/pv/
pviceic/www/test_webdav/dossierA/</D:href>
[..]
<D:couleur-de-cheveux XSI:type="XS:string"><!
[CDATA[blonds]]></D:couleur-de-cheveux>
[..]
</D:response>
[..]
```

Si on veut supprimer cette étiquette ridicule, nous n'avons qu'à exécuter:

```
PROPPATCH /users/p/pv/pviceic/www/test_
webdav/dossierA HTTP/1.1↵
Host: documents.epfl.ch↵
Connection: close↵
Content-Length:154↵
↵
<?xml version="1.0" encoding="utf-8" ?>↵
<D:propertyupdate xmlns:D="DAV:"><D:remove>
<D:prop><D:couleur-de-cheveux/></D:prop></
D:remove></D:propertyupdate>↵
```

Si vous essayez d'exécuter ces derniers exemples par vous-mêmes, le serveur vous retournera le code *401 Forbidden* (Interdit). Comme vous n'avez pas le droit d'écriture sur les dossiers dans l'exemple, vous n'avez pas non plus le droit d'écriture ou modification de ses propriétés. Afin de pouvoir tester par vous mêmes, créez un dossier dans votre propre espace *www* sur documents.epfl.ch via my.epfl et autorisez *Tout public* en lecture, écriture et suppression. Bien entendu, vous remplacerez l'URI qui suit la méthode PROPPATCH par l'URI de votre dossier.

Vous remarquerez que je ne parle pas du tout des éventuelles procédures d'authentification. Je fais cela pour alléger le texte, mon but n'étant pas de vous noyer d'informations, mais de présenter les aspects les plus intéressants du protocole.

L'authentification HTTP est, comme son nom l'indique, *Basic* (en.wikipedia.org/wiki/Basic_access_authentication). Elle est constituée par un en-tête supplémentaire qui fournit au serveur le nom de l'utilisateur et le mot de passe sous une forme légèrement encodée.

Comme my.epfl permet de définir facilement les droits d'accès sur le serveur WebDAV documents.epfl.ch, ne nous privons pas et facilitons-nous la tâche en autorisant - temporairement - tout le monde à modifier les dossiers exemples.

MKCOL

MKCOL (*MaKe COLlection*) est la méthode qui permet de créer les collections - les dossiers.

```
MKCOL /users/p/pv/pviceic/www/test_webdav/
dossierA/sous_dossier/ HTTP/1.1↵
Host: documents.epfl.ch↵
Connection: close↵
```

```
HTTP/1.1 201 Created
[..]
```

En vous rendant dans le dossier en question, vous y verrez le sous-dossier ainsi créé.

DELETE

Comme exercice, je vous laisse découvrir (ou deviner) le résultat de la requête suivante:

```
DELETE /users/p/pv/pviceic/www/test_webdav/
dossierA/sous_dossier/ HTTP/1.1↵
Host: documents.epfl.ch↵
Connection: close↵
```

PUT

La méthode PUT, à vrai dire, figurait déjà dans la spécification originelle (1.0) du protocole HTTP. Elle se trouve toutefois dans les annexes, comme n'étant pas vraiment implémentée par les serveurs. HTTP/1.1 lui donne une place parmi les habituels GET, POST et HEAD¹.

Pour définir la méthode PUT dans le contexte HTTP, je dirais qu'elle *peut* être vue *un peu* comme *une sorte* de POST. Je crois avoir pris assez de précautions en le disant ainsi. Plus précisément, PUT peut avoir comme destination une URI qui n'existe pas - et qui sera créée si tout va bien - tandis que POST agit forcément sur une URI existante. PUT crée ou modifie une ressource, POST envoie un formulaire. PUT est idempotente, POST ne l'est pas. POST permet le téléchargement des fichiers, PUT aussi:

```
PUT /users/p/pv/pviceic/www/test_webdav/
dossierA/titi.txt HTTP/1.1↵
Host: documents.epfl.ch↵
Connection: close↵
Content-Length:23↵
↵
Bien bonjour à vous:)
```

```
HTTP/1.1 201 Created
[...]
```

Dans l'exemple ci-dessus, j'ai exécuté la méthode PUT sur l'URI ../dossierA/titi.txt et j'ai placé un texte dans le corps du message. Comme résultat, j'ai obtenu un fichier titi.txt avec comme contenu le texte `Bien bonjour à vous:)`.

PUT est donc utilisé dans HTTP, comme dans WebDAV, pour l'*upload* des fichiers. La différence est que dans WebDAV c'est la seule possibilité, tandis que dans HTTP on peut télécharger les fichiers avec la méthode POST également. Pourquoi avoir deux méthodes pour l'*upload*? Bien, la vision pragmatique est que c'est plutôt le résultat d'un enchaînement malheureux d'effets de bord coïncidents (suivant l'adage *quand le vin est tiré il faut le boire...*). La version révisionniste explique toutefois que, quand on télécharge un fichier avec la méthode POST, on indique à l'URI spécifiée (celle qui vient après la méthode) qu'on aimerait envoyer un fichier - libre à elle (à l'URI) de décider où le placer. En utilisant PUT, nous disons où nous voulons exactement avoir ce fichier-là. Bien entendu, pour simplifier le tout, le serveur peut décider, à la suite d'une méthode PUT, de refuser la création du fichier à l'URI demandée et de le créer ailleurs. Il doit toutefois l'indiquer en envoyant le code 301 *Moved Permanently* (déplacé de façon permanente), ce qui n'est pas le cas avec la méthode POST.

Quoi qu'il en soit, l'envoi de fichiers est beaucoup plus propre et beaucoup plus simple avec la méthode PUT qu'avec la méthode POST. De plus, comme cette méthode est idempotente, nous pouvons télécharger plusieurs fois un fichier à la même URI, en remplaçant ainsi son contenu (ou en créant une nouvelle version, comme on le verra plus tard, mais alors beaucoup plus tard ☺)

COPY

La méthode COPY sert à copier le fichier

```
COPY /users/p/pv/pviceic/www/test_webdav/
dossierA/titi.txt HTTP/1.1↵
Host: documents.epfl.ch↵
Connection: close↵
Destination: http://documents.epfl.ch/users/p/
pv/pviceic/www/test_webdav/dossierB/titi.
txt↵
```

```
HTTP/1.1 201 Created
[...]
```

... et les dossiers:

```
COPY /users/p/pv/pviceic/www/test_webdav/
dossierA HTTP/1.1↵
Host: documents.epfl.ch↵
Connection: close↵
Destination: http://documents.epfl.ch/users/p/
pv/pviceic/www/test_webdav/dossierB/dossierA↵
```

Si vous suivez les exemples par vous-mêmes, n'oubliez pas de donner les droits d'accès au public sur le dossier **dossierB**.

MOVE

La méthode MOVE sert à déplacer les ressources et à les renommer:

```
MOVE /users/p/pv/pviceic/www/test_webdav/
dossierB/dossierA HTTP/1.1↵
Host: documents.epfl.ch↵
Connection: close↵
Destination: http://documents.epfl.ch/
users/p/pv/pviceic/www/test_webdav/dossierB/
dossierAbis↵
```

Dans cet exemple, dossierB/dossierA est renommé en dossierB/dossierAbis

LOCK/UNLOCK

Les méthodes LOCK et UNLOCK permettent la manipulation des verrous. Les verrous sont les étiquettes d'un genre particulier dont le rôle est d'indiquer que le fichier est édité par quelqu'un. Ainsi, si on essayé de remplacer un fichier verrouillé ou de créer un fichier dans un dossier verrouillé, le système nous en empêche et nous propose des alternatives – comme *attendre le déverrouillage*.

```
LOCK /users/p/pv/pviceic/www/test_webdav/
dossierA/titi.txt HTTP/1.1↵
Host: documents.epfl.ch↵
Connection: close↵
Content-Length:157↵
↵
<?xml version="1.0" encoding="utf-8" ?>↵
<D:lockinfo xmlns:D='DAV:'>>
D:lockscope><D:exclusive/></
D:lockscope><D:locktype><D:write/></
D:locktype></D:lockinfo>↵
```

¹ Bon, vous vous en doutez, je ne vous ai pas TOUT dit sur HTTP/1.1, notamment je ne vous ai pas parlé des méthodes TRACE et CONNECT. Si toutefois vous lisez la norme, vous comprendrez pourquoi elles n'apporteraient rien à cet article.

```
HTTP/1.1 200 OK
[... ]
Lock-Token: <opaque-locktoken:kissrv4.epfl.ch-
LockToken1:72797>
[... ]
<?xml version="1.0" encoding="utf-8" ?>
<D:prop xmlns:D="DAV:">
<D:lockdiscovery>
<D:activelock>
[... ]
<D:locktoken><D:href>opaque-locktoken:kis
srv4.epfl.ch-LockToken1:72797</D:href></
D:locktoken>
</D:activelock>
</D:lockdiscovery>
</D:prop>
```

Si maintenant on essaie d'écrire dans le fichier (avec la méthode PUT), on obtient la réponse:

```
HTTP/1.1 423 Locked↵
[... ]↵
```

J'avoue, là je triche un peu. En fait, si on verrouille un fichier, on devrait pouvoir écrire dedans, ce sont les autres qui doivent en être prévenus. Comme, à titre d'exemple, nous travaillons non authentifiés, le serveur considère le verrou comme étant sans propriétaire et ne nous laisse pas écrire dans le fichier (nous ne sommes pas *Personne*). Nous devrions nous authentifier et ensuite faire les opérations pour avoir un exemple pertinent.

Pour finir, voici comment on déverrouille une ressource:

```
UNLOCK /users/p/pv/pviceic/www/test_webdav/
dossierA/titi.txt HTTP/1.1↵
Host: documents.epfl.ch↵
Connection: close↵
Lock-Token: <opaque-locktoken:kissrv4.epfl.ch-
LockToken1:72797>↵
```

```
HTTP/1.1 204 No Content
[... ]
```

Je tais à dessein beaucoup d'informations sur les verrous WebDAV. Ils peuvent par exemple être exclusifs ou partagés. On peut également leur spécifier une durée au delà de laquelle ils se *déverrouillent*. J'espère toutefois que les deux petits exemples ci-dessus permettront de saisir l'essentiel. Il n'est pas inutile de mentionner également que l'apparente simplicité de verrouillage/déverrouillage des ressources ne rend pas justice à la complexité des algorithmes du côté serveur. Implémenter un système de verrous partagés sur une arborescence n'est en aucun cas une tâche triviale ...

CONCLUSION

Nous voilà arrivés au bout de cet article. Nous avons vu comment utiliser un protocole simple – HTTP – comme fondation de quelque chose de beaucoup plus complexe. D'aucuns diront peut-être que HTTP est un protocole obsolète, car immobile. Pour ma part, *immobile* est un adjectif qui sied à merveille aux fondations.

Dans l'article suivant, nous parlerons des extensions au protocole WebDAV. Ainsi nous verrons, toujours avec les exemples, le fonctionnement de WebDAV Access Control Protocol (RFC 3744) qui permet la gestion des droits d'accès. Nous parlerons également du DELTAV (RFC 3253) qui permet de manipuler les différentes versions d'un document. Pour finir, nous rentrerons dans les détails du DASL *DAV Searching and Locating* (RFC en draft) qui permet d'exécuter les recherches sur les serveurs WebDAV.

Le troisième article, parlera lui du CalDAV, un protocole basé sur WebDAV et qui ouvre la voie à une nouvelle manière de concevoir les agendas en ligne et de les partager. ■