RAPPORT TECHNIQUE

IMPLÉMENTATION DU MODULE WEBDAV POUR LA LIBWWW

Manuele Kirsch Pinheiro

(Manuele.Kirsch_Pinheiro@inrialpes.fr)

1 Introduction

Ce document décrit l'implémentation d'un module cliente WebDAV dans la librairie Libwww. La librairie Libwww est une API pour la création des clients Web. C'est une librairie fortement modulaire, écrite en C pour les architectures Unix et Win32. Parmi ses module standard, la Libwww fournit un support complet au protocole HTTP/1.1 et aussi un analyseur syntaxique pour XML (Expat).

En respectant la structure de la Libwww, nous avons l'introduit un nouveau module pour le support au côté client du protocole WebDAV. Le protocole WebDAV est une extension du protocole HTTP/1.1 qui définit des nouvelles méthodes pour la rédaction éloignée sur le Web.

Dans ce document, nous faisons la description de ce nouveau module en quatre parties. Dans une première partie, nous décrivons le protocole WebDAV et les méthodes qu'il a proposées. En suite, nous présentons le module "HTDAV", en décrivant sa structure interne. Puis, nous présentons les modifications qui ont été réalisées sur la Libwww. Finalement, nous présentons quelques résultats.

2 Le protocole WebDAV

Le protocole WebDAV est une extension du protocole HTTP/1.1 qui définit des nouvelles méthodes pour la rédaction éloignée sur le Web. Il s'agit d'une infrastructure standard pour la rédaction en collaboration asynchrone sur Internet.

La spécification du protocole WebDAV définit sept nouvelles méthodes : PROPFIND et PROPPATH pour la gestion des propriétés, LOCK et UNLOCK pour le verrouillage des ressources, COPY et MOVE pour la gestion de *namespace*, et MKCOL pour la création des collections. En additions à ces méthodes, la spécification WebDAV définit aussi des modifications dans le comportement des méthodes DELETE et PUT du HTTP/1.1, à fin qu'ils puissent traiter les propriétés et les collections. Le protocole WebDAV a aussi défini des nouveaux *headers*, *request* et *response entity body* (tout représentés en XML), lesquels sont utilisés par les nouvelles méthodes.

2.1 Les Nouveaux *Headers*

Le protocole WebDAV a défini huit *headers*: deux pour les réponses du serveur et six pour les réquisitions. Les deux premiers sont le *header* `DAV` et `Status-Uri`. Le *header* `DAV` est utilisé dans la réponse de la méthode OPTIONS pour indiquer si le serveur support le protocole WebDAV. Un serveur est classé "classe `2`" s'il supporte des verrous et toutes les définitions du protocole, ou il est classé "classe `1`" s'il ne supporte que les définitions "MUST" de la description du protocole. Le *header* `Status-Uri` est utilisé dans les réponses du type "`102 Processing`" pour indiquer les résultats (le *status code*) temporaires d'une ressource.

Les autres *headers* sont utilisés dans les réquisitions. Ces *headers* sont les suivants :

- *`If` Header*

Le *header* `If` définit une série des *state tokens*. Les typiques *state tokens* sont les *lock token*, lesquels représentent un verrou. À chaque nouveau verrou créé, le serveur doit retourner, dans le *header* `Lock-Token`, un *lock token* unique pour la ressource verrouillée. Le client pourra, donc, utiliser le *header* `If` avec cette valeur pour faire d'autres réquisitions sur la même ressource. Le format de *header* `If` est le suivant :

```
If = "If" ":" ( 1*No-tag-list | 1*Tagged-list)
No-tag-list = List
Tagged-list = Resource 1*List
Resource = Coded-URL
List = "(" 1*(["Not"](State-token | "[" entity-tag "]")) ")"
State-token = Coded-URL
Coded-URL = "<" absoluteURI ">"
```

- *`Depth` Header*

Le *header* `Depth` est utilisé quand une méthode est applicable sur des collections. Il indique si le serveur doit exécuter la méthode sur la collection seulement (si la valeur est 0), ou sur ses membres directs (si la valeur est 1), ou sur toute sa hiérarchie (valeur "infinity"). Le format de *header* `Depth` est le suivant :

```
Depth = "Depth" ":" ("0" | "1" | "infinity")
```

- *`Lock-Tocken` Header*

Le *header* `Lock-Token` est utilisé par la méthode UNLOCK pour identifier le verrou qui doit être effacé. Le format de *header* `Lock-Token` est le suivant :

```
Lock-Token = "Lock-Token" ":" Coded-URL
```

- *`Destination` Header*

Le *header* `Destination` est utilisé par les méthodes COPY et MOVE pour indiquer la URI de destination. Son format est le suivant :

```
Destination = "Destination" ":" absoluteURI
```

- *`Timeout` Header*

Le *header* `Timeout` est utilisé par le méthode LOCK pour indiquer la valeur désirée pour le temps d'expiration du verrou. Néanmoins, il faut remarquer que le serveur peut ignorer le *header* `Timeout` envoyé par le client, et aussi que le client ne peut pas soumettre ce *header* s'il est en train de demander un rafraîchissement d'un verrou.

```
TimeOut = "Timeout" ":" 1#TimeType
TimeType = ("Second−" DAVTimeOutVal | "Infinite" | Other)
DAVTimeOutVal = 1*digit Other = "Extend" field−value
```

- *`Overwrite` Header*

Le *header* `Overwrite` est utilisé par les méthodes COPY et MOVE pour indiquer si le serveur doit superposer la ressource destination.

```
Overwrite = "Overwrite" ":" ("T" | "F")
```

2.2 Les Nouveaux *Status Code*

Le protocole WebDAV a défini aussi des nouveaux *status codes*. Ils sont les suivants :

- *"102 Processing"* : c'est une réponse intermédiaire utilisée par le serveur pour informer le client qu'il a accepté sa réquisition, mais il n'a pas la finie. Le serveur doit, tant qu'il aura fini la réquisition, envoyer au client une réponse finale.

- *"207 Multi−Status"* : cette réponse fournit au client les status de plusieurs opérations indépendantes d'un seul coup. Dans cette réponse il y a toujours un *entity body* XML avec l'élément `multistatus`. Ceci peut inclure plusieurs éléments `response`, lesquels auront les *status codes* produits pendant l'exécution de la réquisition.

- *"422 Unprocessable Entity*" : ce *status cod*e est utilisé par le serveur pour indiquer au client qu'il a bien compris le contenu et la syntaxe de la réquisition, mais il ne peut pas l'exécuter.

- *"423 Locked"* : cette réponse indique que la ressource, destination ou source, est verrouillée.

- *"424 Failed Dependency"* : ce *status cod*e indique que la méthode ne peut pas être exécutée, car elle dépend d'une autre action qui a échoué.

- *"507 Insufficient Storage"* : ceci est utilisé par le serveur pour indiquer qu'il n'est pas capable de sauvegarder la représentation nécessaire à l'exécution de la méthode.

2.3 Les Nouveaux Méthodes

2.3.1 PROPPATH et PROPFIND

Les méthodes PROPPATH et PROPFIND ont été conçu pour la manipulation des propriétés. Les propriétés sont des morceaux d'information qui décrivent l'état d'une ressource. Elle sont des meta−donnés formés par la paire nom /valeur. Le nom est un URI qui fait référence à la syntaxe et à la sémantique de la propriété, et la valeur est un code XML bien formé (*well−formed*) .

La méthode PROPFIND demande au serveur les propriétés définies d'une ressource, identifiée par le *header* `Request−URI,` et les propriétés de tous ces membres, si cette ressource est une collection. Le client peut demander, dans le *request entity body*, quelques propriétés en particulier (à travers des éléments XML `propfind` et `prop`), toutes les propriétés de la ressource (élément XML `allprop`) ou une liste avec tous les noms des propriétés de la ressource (élément `propname`). Il peut aussi, en contrôlant la valeur de *header* `Depth`, demander les propriétés de tous les membres d'une collection, ou seulement des membres directs, ou encore les propriétés de la collection elle−même. Le Cadre 1 présente un exemple de réquisition PROPFIND.

Request :

```
PROPFIND /upload/bind.html HTTP/1.1
Accept: text/xml
TE: trailers
Host: gnome:1959
User−Agent: MyDAV/2.0 libwww/5.3.2
Cache−Control: no−cache
Connection: TE,Keep−Alive
Pragma: no−cache
Content−Length: 134
Content−Type:text/xml

<?xml version='1.0' encoding='utf−8'?>
<D:propfind xmlns:D='DAV:'>
  <D:prop>
     <D:supportedlock />
  </D:prop>
</D:propfind>
```

Response :

```
HTTP/1.1 207 Multi−Status
Date: Wed, 27 Feb 2002 14:46:08 GMT
Server: Apache/1.3.22 (Unix) DAV/1.0.2
Keep−Alive: timeout=15, max=100
Connection: Keep−Alive
Transfer−Encoding: chunked
Content−Type: text/xml; charset="utf−8"

<?xml version="1.0" encoding="utf−8"?>
 <D:multistatus xmlns:D="DAV:">
  <D:response>
   <D:href>/upload/bind.html</D:href>
   <D:propstat>
    <D:prop>
     <D:supportedlock>
      <D:lockentry>
        <D:lockscope>
          <D:exclusive/>
        </D:lockscope>
        <D:locktype>
          <D:write/>
        </D:locktype>
      </D:lockentry>
      <D:lockentry>
        <D:lockscope>
          <D:shared/>
        </D:lockscope>
        <D:locktype>
          <D:write/>
        </D:locktype>
      </D:lockentry>
     </D:supportedlock>
    </D:prop>
    <D:status>HTTP/1.1 200 OK</D:status>
   </D:propstat>
  </D:response>
 </D:multistatus>
```

Cadre 1. *Une réquisition PROPFIND. Dans cet exemple, le client demande la propriété "supportedlock" au serveur. Celui−ci lui répond avec un response entity body en XML, en décrivant la propriété "supportedlock" de la ressource.*

La méthode PROPPATH est similaire à la méthode PROPFIND, sauf qu'il s'agit d'une méthode pour définir ou effacer des propriétés. Le client utilise le *request entity body* pour spécifier les opérations qu'il voudrait (élément `set` pour

définir ou changer, et élément `remove` pour effacer), et les propriétés sur lesquelles ces opérations doivent être exécutées. Le Cadre 2 présente un exemple de réquisition PROPPATH.

Request :

```
PROPPATCH /upload/dso.html HTTP/1.1
Accept: text/xml
TE: trailers
Host: gnome.inrialpes.fr:1959
User-Agent: MyDAV/2.0 libwww/5.3.2
Cache-Control: no-cache
Connection: TE,Keep-Alive
Pragma: no-cache
If: (<opaquelocktoken:2850c7f2-1dd2-11b2-
b731-d63519929aa3>)
Content-Length: 214
Content-Type: text/xml

<?xml version='1.0' encoding='utf-8'?>
<D:propertyupdate xmlns:D='DAV:'>
 <D:set>
  <D:prop xmlns:A='awareness.dtd'>
    <A:info> Apache Manual - Dynamic
shared object </A:info>
  </D:prop>
 </D:set>
</D:propertyupdate>
```

Response :

```
HTTP/1.1 207 Multi-Status
Date: Wed, 27 Feb 2002 16:09:00 GMT
Server: Apache/1.3.22 (Unix) DAV/1.0.2
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/xml; charset="utf-8"

<?xml version="1.0" encoding="utf-8"?>
<D:multistatus xmlns:D="DAV:"
xmlns:ns1="awareness.dtd"
xmlns:ns0="DAV:">
 <D:response>
  <D:href>/upload/dso.html</D:href>
  <D:propstat>
    <D:prop>
      <ns1:info/>
    </D:prop>
    <D:status>HTTP/1.1 200 OK</D:status>
  </D:propstat>
 </D:response>
</D:multistatus>
```

Cadre 2. *Un exemple de réquisition PROPPATCH, où le client définit la propriété "info" pour la ressource. Il faut observer l'utilisation d'autres namespaces, non le namespace DAV, pour les propriétés.*

2.3.2 MKCOL

La méthode MKCOL crée une nouvelle collection dans le URI spécifié dans le *header* `Request-URI`. Une collection est un type de ressource Web, similaire à un répertoire, dont l'état est, au minimum, une liste de URIs membres internes et d'un ensemble des propriétés. Cet URI identifié dans le *header* `Request-URI` ne peut pas exister avant l'appel de la méthode MKCOL, mais tous ces ancêtres doivent exister avant, autrement la méthode ira échouer. Le Cadre 3 présente un exemple de réquisition.

Request :

```
MKCOL /upload/Tst/t1 HTTP/1.1
Accept: text/xml
TE: trailers
Host: gnome.inrialpes.fr:1959
User-Agent: MyDAV/2.0 libwww/5.3.2
Cache-Control: no-cache
Connection: TE,Keep-Alive
Pragma: no-cache
```

Response :

```
HTTP/1.1 201 Created
Date: Wed, 27 Feb 2002 16:15:42 GMT
Server: Apache/1.3.22 (Unix) DAV/1.0.2
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML
2.0//EN">
<HTML>
 <HEAD>
  <TITLE>201 Created</TITLE>
 </HEAD>
 <BODY>
   <H1>Created</H1>
   Collection /upload/Tst/t1 has been created.
   <P><HR>
   <ADDRESS>Apache/1.3.22 Server at
gnome.inrialpes.fr Port 1959</ADDRESS>
 </BODY>
</HTML>
```

Cadre 3. *Création d'une nouvelle collection.*

2.3.3 COPY et MOVE

Les méthodes COPY et MOVE manipulent le *namespace* dans un serveur Web, en copiant ou déplaçant des ressources. La méthode COPY crée un duplicata de la ressource source, identifiée par le *header* `Request-URI`, dans la ressource destination, indiquée par le *header* `Destination`. Le serveur essaye de faire la meilleure copie de la ressource source et ces propriétés sur la ressource destination. Il peut copier toutes les propriétés de la ressource ou seulement une liste d'elles, selon l'élément XML `propertybehavior` dans le *request entity body*. Cette méthode peut aussi copier toute une collection et ses membres, selon le *header* `Depth`. Si une erreur se produit pendant la méthode COPY, le serveur ne doit pas copier les éléments de la hiérarchie où l'erreur a été produite, mais il doit essayer de copier le maximum de la ressource source possible. Le cadre 4 présente un exemple de réquisition.

Request :

```
COPY /upload/Tst/t2 HTTP/1.1
Accept: text/xml
TE: trailers
Host: gnome.inrialpes.fr:1959
User-Agent: MyDAV/2.0 libwww/5.3.2
Cache-Control: no-cache
Connection: TE,Keep-Alive
Pragma: no-cache
Destination:
http://gnome.inrialpes.fr:1959/upload/Tst
/t1
```

Response :

```
HTTP/1.1 423 Locked
Date: Wed, 27 Feb 2002 16:30:38 GMT
Server: Apache/1.3.22 (Unix) DAV/1.0.2
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-
1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML
2.0//EN">
<HTML>
 <HEAD>
   <TITLE>423 Locked</TITLE>
 </HEAD>
 <BODY>
   <H1>Locked</H1>
The requested resource is currently
locked.
The lock must be released or proper
identification
given before the method can be applied.
  <HR>
  <ADDRESS>Apache/1.3.22 Server at
gnome.inrialpes.fr Port 1959</ADDRESS>
 </BODY>
</HTML>
```

Cadre 4. *Un exemple de la méthode COPY, qui a échoué, car la ressource de destination est verrouillée.*

La méthode MOVE est similaire à la méthode COPY, mais il déplace la ressource source vers l'URI destination (identifié par le *header* `Destination`). Son opération est équivalente à une opération de COPY suivie par un processus de soutien de la consistance chez le serveur (il fait des actualisations causées par le mouvement) et par la suppression de la ressource source. Sur les collections, la méthode MOVE doit déplacer tous les membres internes et toute leur hiérarchie (le *header* `Depth` doit absolument être "`infinity`"). Ainsi que la méthode COPY, si une erreur se produit pendant la méthode MOVE, le serveur ne doit pas déplacer les éléments de la hiérarchie où l'erreur a été produite, mais il doit essayer de déplacer le maximum de la ressource source possible. Le cadre 5 présente un exemple de réquisition.

Request :

```
MOVE /upload/Tst/t2 HTTP/1.1
Accept: text/xml
TE: trailers
Host: gnome:1959
User-Agent: MyDAV/2.0 libwww/5.3.2
Cache-Control: no-cache
Connection: TE,Keep-Alive
Pragma: no-cache
Depth: infinity
Overwirte: T
If:
<http://gnome:1959/upload/Tst/t2/sub/>
(<opaquelocktoken:b266c934-1dd1-11b2-
b731-d63519929aa3>)
Destination:
http://gnome:1959/upload/Tst/t1
Content-Length: 130
Content-Type: text/xml

<?xml version='1.0' encoding='utf-8'?>
<D:propertybehavior xmlns:D='DAV:'>
  <D:keepalive>*</D:keepalive>
</D:propertynehavior>
```

Response :

```
HTTP/1.1 204 No Content
Date: Wed, 27 Feb 2002 16:40:36 GMT
Server: Apache/1.3.22 (Unix) DAV/1.0.2
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

Cadre 5. *Une réquisition MOVE réussie. Il faut remarquer l'utilisation des headers If, Depth et Overwrite. Il faut aussi observer l'utilisation d'un request entity body, qui indique au serveur quoi faire avec les propriétés.*

2.3.4 LOCK et UNLOCK

Les méthodes LOCK et UNLOCK sont responsables pour les opérations pour le verrouillage des ressources. Le protocole WebDAV a défini que les verrous peuvent être exclusif (*exclusive lock*) ou partagé (*shared lock).* Le premier garantit que seulement une personne aura le droit d'accès, et le deuxième permet que plusieurs personnes reçoivent le verrou et, par conséquent, le droit d'accès. Le protocole a défini aussi un seul type de verrou pour le droit d'écriture sur une ressource (*write lock*). Alors, il est toujours possible lire le contenu et les propriétés dans ressource. Par contre, seulement quelqu'un avec le verrou pourra exécuter les opérations d'écriture, comme PUT, POST, PROPPATH, LOCK, UNLOCK, MOVE, DELETE et MKCOL. Enfin, s'il s'agit d'un verrou sur une collection, il préviendra aussi l'addition et la suppression des ressources membres pour des personnes qui ne possèdent pas le verrou.

La méthode LOCK crée, ou rafraîchit, un verrou sur la ressource identifiée par le `Request-URI`. Pour créer un verrou, le client doit soumettre des informations sur quel type de verrou il voulait. Ces informations sont insérées dans le *request entity body* de la réquisition, en utilisant l'élément XML `lockinfo`. L'élément `lockinfo` indique si le verrou est exclusif ou partagé (sous-élément `lockscope`) et quel est son type (sous-élément `locktype`). Il doit indiquer aussi le propriétaire du verrou, par l'élément `owner`, et le temps d'expiration du verrou par l'élément `timeout`.

Un verrou possède un temps d'expiration (*timeout*) et un client ne doit jamais soumettre le même verrou deux fois. Par contre, il peut demander un rafraîchissement du verrou, en utilisant la méthode LOCK avec le *header* `If` et sans un *request entity body*. Le serveur doit, dans la première demande du verrou, retourner le temps d'expiration dans le *header* `Timeout`, et il peut aussi, dans une demande de rafraîchissement, retourner ce même *header.* En addition, le client peut soumettre une valeur pour le temps d'expiration dans le *header* `Timeout` de la réquisition, mais le serveur peut ignorer cette valeur. Néanmoins, tous les clients doivent assumer que ses verrous peuvent disparaître à n'importe quand, quoique soit la valeur du temps d'expiration. Le temps qui le serveur retours dans le *header* `Timeout` indique le

comportement du serveur si aucune situation extraordinaire arrive. Les Cadres 6 e 7 présentent des exemples de réquisition LOCK.

Request :

```
LOCK /upload/Manu/awareness.dtd HTTP/1.1
Accept: text/xml
TE: trailers
Authorization: Basic a2lyc2NoOnBhc3N3b3Jk
Host: gnome.inrialpes.fr:1959
User-Agent: MyDAV/2.0 libwww/5.3.2
Cache-Control: no-cache
Connection: TE,Keep-Alive
Pragma: no-cache
Timeout: Second-7200
Content-Length: 272
Content-Type: text/xml

<?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:owner>
    <D:href>
mailto:Manuele.Kirsch_Pinheiro@inrialpes.fr
    </D:href>
  </D:owner>
</D:lockinfo>
```

Response :

```
HTTP/1.1 200 OK
Date: Wed, 27 Feb 2002 16:53:31 GMT
Server: Apache/1.3.22 (Unix) DAV/1.0.2
Lock-Token: <opaquelocktoken:890b21ae-
1dd2-11b2-a9e1-9a092fd83156>
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/xml; charset="utf-8"

<?xml version="1.0" encoding="utf-8"?>
<D:prop xmlns:D="DAV:">
 <D:lockdiscovery>
  <D:activelock>
   <D:locktype><D:write/></D:locktype>
   <D:lockscope> <D:exclusive/>
   </D:lockscope>
   <D:depth>infinity</D:depth>
   <ns0:owner xmlns:ns0="DAV:">
     <ns0:href>
mailto:Manuele.Kirsch_Pinheiro@inrialpes
.fr
     </ns0:href>
   </ns0:owner>
   <D:timeout>Second-7200</D:timeout>
   <D:locktoken>
     <D:href>opaquelocktoken:890b21ae-
1dd2-11b2-a9e1-9a092fd83156</D:href>
   </D:locktoken>
  </D:activelock>
 </D:lockdiscovery>
</D:prop>
```

Cadre 6. *Une réquisition LOCK. Le client demande dans cette réquisition la création d'un nouveau verrou sur la ressource. Il faut remarquer dans la réquisition la présence du request entity body en décrivant le verrou (un verrou d'écriture exclusif) et son propriétaire. Dans la réponse, il faut remarquer la présence du header Lock-Token, lequel possède le state token qui identifie d'une façon unique le verrou créé.*

Request :

```
LOCK /upload/Manu/awareness.dtd HTTP/1.1
Accept: text/xml
TE: trailers
Authorization: Basic
a2lyc2NoOnBhc3N3b3Jk
Expect: 100-continue
Host: gnome.inrialpes.fr:1959
User-Agent: MyDAV/2.0 libwww/5.3.2
Cache-Control: no-cache
Connection: TE,Keep-Alive
Pragma: no-cache
Timeout: Second-7200
If: (<opaquelocktoken:890b21ae-1dd2-
11b2-a9e1-9a092fd83156>)
Transfer-Encoding: chunked
Content-Type: application/octet-stream
```

Response :

```
HTTP/1.1 200 OK
Date: Wed, 27 Feb 2002 17:01:07 GMT
Server: Apache/1.3.22 (Unix) DAV/1.0.2
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/xml; charset="utf-8"

<?xml version="1.0" encoding="utf-8"?>
<D:prop xmlns:D="DAV:">
 <D:lockdiscovery>
  <D:activelock>
   <D:locktype><D:write/></D:locktype>
   <D:lockscope> <D:exclusive/>
   </D:lockscope>
   <D:depth>infinity</D:depth>
   <ns0:owner xmlns:ns0="DAV:">
     <ns0:href>
mailto:Manuele.Kirsch_Pinheiro@inrialpes.f
r
     </ns0:href>
   </ns0:owner>
   <D:timeout>Second-7200</D:timeout>
   <D:locktoken>
    <D:href>opaquelocktoken:890b21ae-1dd2-
11b2-a9e1-9a092fd83156</D:href>
   </D:locktoken>
  </D:activelock>
 </D:lockdiscovery>
</D:prop>
```

Cadre 7. *Une réquisition de rafraîchissement d'un verrou. Le client utilise la méthode LOCK sans le request entity body pour rafraîchir le verrou, en identifiant ceci par le state token dans le header If. Il faut aussi observer que la réponse ne possède pas le header Lock-Token, car c'était une réquisition de rafraîchissement.*

La méthode UNLOCK supprime le verrou identifié par le *header* `Lock−Token`. Cette valeur est une identification unique qui est retournée dans chaque appel réussite de la méthode LOCK. La méthode UNLOCK instruit le serveur à effacer le verrou sur la ressource indiquée (`Request−URI`) et sur toutes les ressources incluses dans le verrou, plutôt dans le cas où la ressource est une collection. Par contre, si toutes ces ressources incluses ne peuvent pas être libérées, la méthode UNLOCK doit échouer. Le cadre 8 présente un exemple de réquisition.

Request :	**Response :**
UNLOCK /upload/Manu/awareness.dtd HTTP/1.1 Accept: text/xml TE: trailers Authorization: Basic a2lyc2NoOnBhc3N3b3Jk Host: gnome.inrialpes.fr:1959 User−Agent: MyDAV/2.0 libwww/5.3.2 Cache−Control: no−cache Connection: TE,Keep−Alive Pragma: no−cache Lock−Token: <opaquelocktoken:890b21ae−1dd2−11b2−a9e1−9a092fd83156>	HTTP/1.1 204 No Content Date: Wed, 27 Feb 2002 17:05:04 GMT Server: Apache/1.3.22 (Unix) DAV/1.0.2 Keep−Alive: timeout=15, max=99 Connection: Keep−Alive Content−Type: text/plain

Cadre 8. *Une réquisition UNLOCK. Le client identifie le verrou qu'il voudrait effacer par le header Lock−Token, et aucun request entity body est envoyé.*

En dehors les méthodes LOCK et UNLOCK, un client DAV peut utiliser la méthode PROPFIND pour découvrir si un serveur supporte des verrous et quels types à travers la propriété `supportedlock`, ce qui s'appelle "*lock capability discovery*". Le *lock capability discovery* est très importante, car le protocole affirme qu'un serveur n'est pas obligé de supporter des verrous. Alors, un client peu l'utilise pour découvrir si un serveur supporte un certain type de verrou. Le client peut aussi, en utilisant la même méthode et la propriété `lockdiscovery`, découvrir quelles sont les ressources verrouillées et qui les possède (*active lock discovery*). C'est important remarquer que le protocole WebDAV ne définit pas, a priori, un contrôle d'accès sur les verrous. Pour ça, il faut utiliser les fonctions d'authentification disponible dans le protocole HTTP/1.1 et supporter par le serveur.

Finalement, il faut remarquer que le protocole n'est pas une garantie que le problème de mise à jour perdue n'ira jamais se produire. Il faut considérer, par exemple, un client A, qui ne supporte pas le protocole WebDAV (un client HTTP seulement), et client B, qui supporte le protocole WebDAV. Le client A peut faire un GET dans une ressource et la changer. Pendant cette période, le client B peut faire un LOCK, suivi par un GET, et aussi change la ressource. Le client B peut terminer ses modifications avant le client A, et il fera un PUT suivi par un UNLOCK. Après tous ça, le client A fera un PUT, en déposant sa nouvelle version de la ressource, et les modifications réalisées par le client B seront perdues (voici la Figure 1 au−dessous). Ce problème se produit parce que le protocole WebDAV est compatible avec des clients HTTP qui ne le comprennent pas et aussi avec des clients et des serveurs qui n'utilisent pas des verrous. Le protocole ne peut pas aussi forcer la séquence LOCK/GET/PUT/UNLOCK, laquelle peut prévenir la mise à jour perdue.

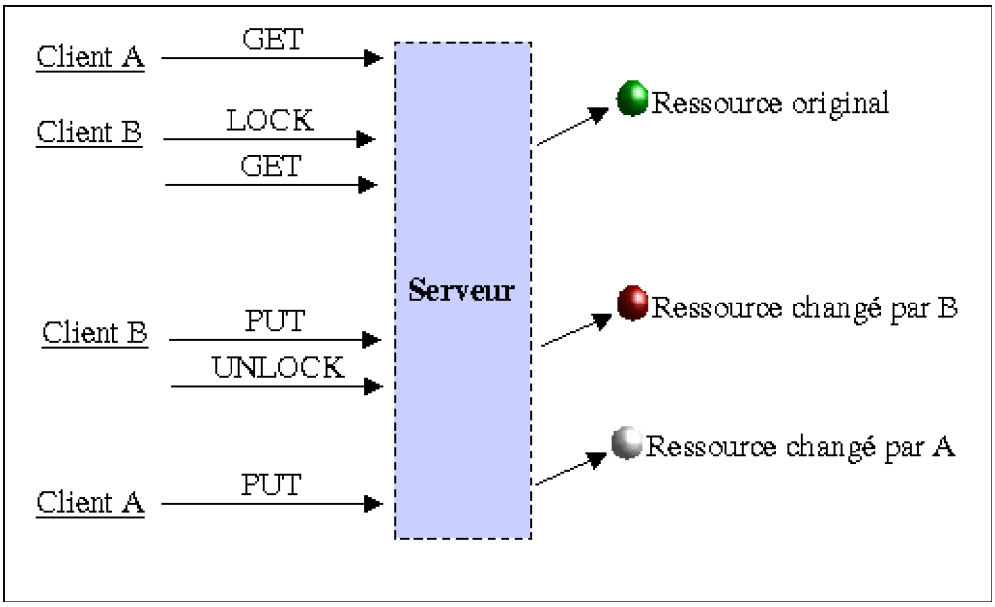

Figure 1. *Comme le problème de mise à jour perdue peut se produire même dans un environment où le protocole WebDAV est disponible.*

3 Le module HTDAV

La librairie Libwww est une API pour la création des clients Web. Elle fournit une infrastructure pour l'accès aux plusieurs protocoles, comme FTP, NNTP, et HTTP/1.1. Il y a aujourd'hui plusieurs applications (par exemple, Amaya), qui utilisent la Libwww pour accéder à ses protocoles. Sa structure flexible permet aux ces que développent ces applications d'introduire dans la Libwww des nouvelles modules, des nouveaux protocoles.

Alors, en profitant de cette flexibilité de la Libwww, nous avons créé un nouveau module pour le protocole WebDAV. L'objectif était d'isoler toutes les fonctions nécessaires à ce protocole dans un seul module, d'une telle façon que les applications qui ne l'utilisent pas ne seraient pas affectées par ce nouveau module. Malheureusement, tandis que le WebDAV n'est pas un protocole indépendant, mais une extension du protocole HTTP, un tel niveau d'isolement n'était pas possible. Alors, nous avons limité les modifications dans la Libwww à ceux qui étaient vraiment nécessaires, et nous avons isolé tout le reste dans un seul module, le module HTDAV. Les modifications que nous avons faites dans la Libwww seront présentées dans la prochaine section, tandis que le nouveau module est présenté ici.

Dans le module HTDAV, nous avons mis les fonctions de haut niveau et les structures nécessaires à ses fonctions. Les fonctions suivent plus ou moins le style des fonctions définies dans le module HTAccess, où se trouvent les fonctions de haut niveau du protocole HTTP. Pour chaque méthode, nous avons défini au minimum des trois fonctions : une pour l'exécution de la méthode sur une URI absolue, une pour les URI relatives, et une méthode pour l'exécution de la méthode sur un objet `HTAnchor`. En réalité, il n'y a qu'une seule méthode qui exécute réellement la méthode (en général, la méthode qui utilise l'`Anchor`). Tous les autres font appel à ceci. D'ailleurs, cette méthode principale utilise le plus possible les fonctions déjà disponibles dans la Libwww. Cette fonction principale fait aussi des petits contrôles dans la réquisition. Par exemple, si la méthode ne permet pas un certain *header*, la

fonction ne lui inclut pas dans la réquisition qui sera envoyé au serveur, même si l'application l'a demandé.

En dehors ces fonctions de haut niveau, le module HTDAV définit aussi une structure, qui est utilisé par toutes ses fonctions. Cette structure, appelée `HTDAVHeaders`, isole tout le *header* possible pour les réquisitions DAV. Elle permet aux applications définir les *headers* DAV qu'elles veulent dans un seul objet et après les utiliser dans les fonctions de haut niveau. La création de cette structure nous a permis de décrire les *headers* DAV et les enclore dans un seul endroit, sans, pour autant, changer la définition de l'objet `HTRequest` (qui représente la réquisition).

Définition interne (fichier HTDAV.c) :

```
struct _HTDAVHeaders {
    char * If;
    char * Depth;
    char * Destination;
    char * LockToken;
    char * Timeout;
    char Overwrite;
};
```

Définition public (fichier HTDAV.h) :

```
typedef struct _HTDAVHeaders HTDAVHeaders;
```

Fonctions publics (fichier HTDAV.h):

```
extern HTDAVHeaders * HTDAVHeaders_new (void);
extern BOOL HTDAVHeaders_delete (HTDAVHeaders *me);
```

Cadre 9. Définition du type HTDAVHeaders et ses fonctions pour créer et détruire des objets.

La définition interne de la structure `HTDAVHeaders` (voir le Cadre 9) est cachée de l'application. Elle ne peut manipuler un objet de cette structure que pour des fonctions spécifiques. Ces fonctions lui permettent de créer, effacer et définir des *headers* DAV dans un objet `HTDAVHeaders`. Elles constituent le seul moyen d'accès à cette structure, hors le module HTDAV. Le Cadre 10 présente les fonctions définies pour la manipulation de chacun des *headers* DAV.

If	extern BOOL **HTDAV_setIfHeader** (HTDAVHeaders *me, char *If); extern BOOL **HTDAV_deleteIfHeader** (HTDAVHeaders * me); extern char * **HTDAV_ifHeader** (HTDAVHeaders *me);
Depth	extern BOOL **HTDAV_setDepthHeade**r (HTDAVHeaders *me, char *Depth); extern BOOL **HTDAV_deleteDepthHeader** (HTDAVHeaders * me); extern char * **HTDAV_DepthHeader** (HTDAVHeaders *me);
Lock-Token	extern BOOL **HTDAV_setLockTokenHeader** (HTDAVHeaders *me, char *LockToken); extern BOOL **HTDAV_deleteLockTokenHeader** (HTDAVHeaders * me); extern char * **HTDAV_LockTokenHeader** (HTDAVHeaders *me);
Destination	extern BOOL **HTDAV_setDestinationHeader** (HTDAVHeaders *me, char *Destination); extern BOOL **HTDAV_deleteDestinationHeader** (HTDAVHeaders * me); extern char * **HTDAV_DestinationHeader** (HTDAVHeaders *me);
Timeout	extern BOOL **HTDAV_setTimeoutHeader** (HTDAVHeaders *me, char *Timeout); extern BOOL **HTDAV_deleteTimeoutHeader** (HTDAVHeaders * me); extern char * **HTDAV_TimeoutHeader** (HTDAVHeaders *me);
Overwrite	extern BOOL **HTDAV_setOverwriteHeader** (HTDAVHeaders *me, BOOL Overwrite); extern BOOL **HTDAV_deleteOverwriteHeader** (HTDAVHeaders * me); extern BOOL **HTDAV_OverwriteHeader** (HTDAVHeaders * me);

Cadre 10. Les fonctions pour définir, effacer et récupérer chacun de headers DAV.

Le module HTDAV propose trois fonctions publiques pour chaque méthode du protocole WebDAV. Ces fonctions reçoivent plusieurs paramètres, comme l'objet réquisition, l'URI destination et un XML *body*, si nécessaire, mais aussi un objet `HTDAVHeaders` avec les *headers* que l'application voudrait. En possession de toutes ces informations, chaque fonction essayera de remplir la réquisition et la soumettre à la Libwww. Au−dessous il y a une description des ces fonction pour chaque méthode.

- *Méthode LOCK*

Les fonctions pour la méthode LOCK prennent un *request entity body* en XML, quand il est disponible, et elles prennent aussi de l'objet `HTDAVHeaders` les headers `If`, `Depth` et `Timeout`. Dehors ces *headers* spécifiques pour le WebDAV, il y a aussi quelques *headers* HTTP utilisés pour éviter que les résultats de la réquisition soient gardés dans une *cache*.

Fonction principale :

```
extern BOOL HTLOCKDocumentAnchor (HTRequest *
       request, HTAnchor * dst, HTParentAnchor
       *xmlbody, HTDAVHeaders *headers);
```

Autres fonctions :

```
extern BOOL HTLOCKAnchor (HTRequest *
       request,HTAnchor * dst, char * xmlbody,
       HTDAVHeaders * headers);

extern BOOL HTLOCKAbsolute (HTRequest * request,
       const char * uri, char * xmlbody,
       HTDAVHeaders * headers);

extern BOOL HTLOCKRelative (HTRequest * request,
       const char * relative, HTParentAnchor *
       base, char * xmlbody, HTDAVHeaders *
       headers);
```

- *Méthode UNLOCK*

La méthode UNLOCK ne prend jamais de *message body*, mais elle oblige toujours la présence de *header* Lock−Token, ce qui veut dire que la présence d'un objet `HTDAVHeaders` valide est nécessaire, si non la fonction va échouer.

Fonction principale :

```
extern BOOL HTUNLOCKAnchor (HTRequest * request,
       HTAnchor * dst, HTDAVHeaders * headers);
```

Autres fonctions :

```
extern BOOL HTUNLOCKAbsolute (HTRequest *
       request, const char * uri, HTDAVHeaders *
       headers);

extern BOOL HTUNLOCKRelative (HTRequest *
       request, const char * relative,
       HTParentAnchor * base, HTDAVHeaders *
       headers);
```

- *Méthode PROPFIND*

L'utilisation d'un *message body* en XML dans les réquisitions PROPFIND est facultative. Alors, l'application peut soumettre un objet nul (non−valide) comme *message body*. Le même se produit avec les *headers*. L'application peut soumettre un objet `HTDAVHeaders` avec le *header* `Depth` (valeurs "`0`", "`1`" ou "`infinity`"), ou encore, soumettre un objet nul.

Fonction principale :

```
extern BOOL HTPROPFINDAnchor (HTRequest *
       request, HTAnchor * dst, const char *
       xmlbody, HTDAVHeaders * headers);
```

Autres fonctions :

```
extern BOOL HTPROPFINDDocumentAnchor (HTRequest
       * request, HTAnchor * dst, HTParentAnchor
       * xmlbody, HTDAVHeaders * headers);

extern BOOL HTPROPFINDAbsolute (HTRequest *
       request, const char * uri, const char *
       xmlbody, HTDAVHeaders * headers);

extern BOOL HTPROPFINDRelative (HTRequest *
       request, const char * relative,
       HTParentAnchor * base, const char *
       xmlbody, HTDAVHeaders * headers);
```

- *Méthode PROPPATCH*

Pour les réquisitions PROPPATCH, l'utilisation d'un *message body* en XML est obligatoire. Alors, l'application doit soumettre un objet valide dans le paramètre `xmlbody`. Dans les *headers*, seulement le `If` peut être utilisé dans ce type de réquisition. Si l'application passe aux fonctions un objet `HTDAVHeaders` avec d'autres *headers*, les fonctions iront les ignorer au moment de remplir la réquisition.

Fonction principale :

```
extern BOOL HTPROPPATCHAnchor (HTRequest *
       request, HTAnchor * dst, const char *
       xmlbody,HTDAVHeaders * headers);
```

Autres fonctions :

```
extern BOOL HTPROPPATCHDocumentAnchor (HTRequest
       * request,HTAnchor * dst,  HTParentAnchor
       * xmlbody,HTDAVHeaders * headers);

extern BOOL HTPROPPATCHAbsolute (HTRequest *
       request, const char * uri,  const char *
       xmlbody, HTDAVHeaders * headers);

extern BOOL HTPROPPATCHRelative (HTRequest *
       request, const char * relative,
       HTParentAnchor * base, const char *
       xmlbody, HTDAVHeaders * headers);
```

- *Méthode MKCOL*

Pour les fonctions qui exécutent la méthode MKCOL, aucun *message body* est fourni, et la présence des *headers* est facultative et limitée au *header* `If`.

Fonction principale :

```
extern BOOL HTMKCOLAnchor (HTRequest * request,
       HTAnchor * dst, HTDAVHeaders * headers);
```

Autres fonctions :

```
extern BOOL HTMKCOLAbsolute (HTRequest *
       request, const char * uri, HTDAVHeaders *
       headers);
extern BOOL HTMKCOLRelative (HTRequest *
       request, const char * relative,
       HTParentAnchor * base, HTDAVHeaders *
       headers);
```

- *Méthode COPY*

Les fonctions qui exécutent la méthode COPY obligent la présence d'un objet `HTDAVHeaders`, car le *header* `Destination` doit être défini pour qui la méthode réussit. Les *headers* `If` et `Depth` sont aussi possibles, et la présence d'un *message body* est facultative.

Fonction principale :

```
extern BOOL HTCOPYAnchor (HTRequest *request,
       HTAnchor * src, const char * xmlbody,
       HTDAVHeaders * headers);
```

Autres fonctions :

```
extern BOOL HTCOPYDocumentAnchor (HTRequest *
       request, HTAnchor * src, HTParentAnchor *
       xmlbody, HTDAVHeaders * headers);

extern BOOL HTCOPYAbsolute (HTRequest * request,
       const char * uri, const char * xmlbody,
       HTDAVHeaders * headers);

extern BOOL HTCOPYRelative (HTRequest * request,
       const char * relative, HTParentAnchor *
       base, const char * xmlbody, HTDAVHeaders *
       headers);
```

- *Méthode MOVE*

Les fonctions qui exécutent la méthode MOVE suivent les mêmes restrictions que les fonctions de la méthode COPY.

Fonction principale :

```
extern BOOL HTMOVEAnchor (HTRequest * request,
       HTAnchor * src, const char * xmlbody,
       HTDAVHeaders * headers);
```

Autres fonctions :

```
extern BOOL HTMOVEDocumentAnchor (HTRequest *
       request, HTAnchor * src, HTParentAnchor *
       xmlbody, HTDAVHeaders * headers);

extern BOOL HTMOVEAbsolute (HTRequest * request,
       const char * uri, const char * xmlbody,
       HTDAVHeaders * headers);

extern BOOL HTMOVERelative (HTRequest * request,
       const char * relative, HTParentAnchor *
       base, const char * xmlbody, HTDAVHeaders *
       headers);
```

4 Les modifications dans la Libwww

Pour intégrer ce module HTDAV dans la Libwww, nous avons dû changer quelques modules liés au protocole HTTP, car le WebDAV n'est pas vraiment un protocole indépendant, mais une extension du protocole HTTP. Ces modifications sont liés aux nouveaux *status codes* et aux nouveaux méthodes définis par le WebDAV. Il faudrait définir dans la Libwww ces nouveaux méthodes et introduire les *status codes* dans la machine d'états utilisé dans la librairie pour gérer les réquisitions HTTP. Pour cela, nous avons modifié le module HTTP lui−même, la définition des méthodes (HTMethod), et les définitions de *error codes* et *return codes,* dans les modules HTUtils et HTError. Le Table 1 détaille ces modifications.

Table 1. *Modifications réalisées dans la Libwww. Tous ses modifications ont été bien signalé dans le code source par une directive (HT_DAV). Seulement quand cette directive est définie, les modifications ont son effet dans la librairie.*

Module	***Description des modifications***
HTError	Définition de nouveaux codes d'erreur pour les situations produites par les *status codes 422, 423, 424* et *507* : • "*422 Unprocessable*" − code d'erreur "`HTERR_UNPROCESSABLE`" • "*423 Locked*" − code d'erreur "`HTERR_LOCKED`" • "*424 Failed Dependency*" − code d'erreur "`HTERR_FAILED_DEPENDENCY`" • "*507 Insufficient Storage*" − code d'erreur "`HTERR_INSUFFICIENT_STORAGE`"
HTUtils	Définition des codes de retour possible dans WebDAV : • `HT_PROCESSING` = *102 Processing* • `HT_MULTI_STATUS` = *207 Multi−Status* • `HT_UNPROCESSABLE` = *−422 Unprocessable* • `HT_LOCKED` = *−423 Locked* • `HT_FAILED_DEPENDENCY` = *−424 Failed Dependency* • `HT_INSUFFICIENT_STORAGE` = *−507 Insufficient Storage*
HTMethod	Définition des nouvelles méthodes WebDAV dans la structure HTMethod. Cette structure est un *bitset*, laquelle identifie les méthodes qui peuvent être utilisées par les réquisitions dans la Libwww. Le module HTMethod fait aussi la conversion entre cette représentation (*bitset*) et le nom de la méthode, utilisé dans la réquisition. L'implémentation de ces méthodes elle−même a été faite dans le module HTDAV. • `METHOD_LOCK = 0x400,` • `METHOD_UNLOCK = 0x800,` • `METHOD_PROPFIND = 0x1000,` • `METHOD_PROPPATCH = 0x2000,` • `METHOD_MKCOL = 0x4000,` • `METHOD_COPY = 0x8000,` • `METHOD_MOVE = 0x10000,`
HTTP	L'implémentation du protocole HTTP dans la Libwww travail avec une machine d'état. Les nouveaux *status codes* définis ont été introduit dans cette machine, afin que le module HTTP puisse les reconnaître. Les fonctions affectées dans le code source du module étaient `HTTPInformation`, `HTTPNextState` et `stream_pipe`.

Plusieurs de ces modifications ont été présentées dans la liste de discussion de la librairie Libwww. Dans cette même liste, un point important sur la capacité d'expansion de la librairie a été aussi présenté. Le fait est que le protocole WebDAV

n’est pas la seule proposition d’expansion du protocole HTTP. Il y a aussi d’autres propositions liées à ceci. Il y a, par exemple, les propositions sur le versionnement proposées par le groupe IETF DELTA−V, ou encore les extensions pour le contrôle d’accès sur des ressources DAV proposées par le groupe ACL (un subgroupe du groupe IETF WebDAV). Alors, une fois que quelqu’un désire utiliser la Libwww avec une de ces extensions du protocole HTTP, il lui faudrait changer la librairie, introduire les nouvelles méthodes, etc. Ceci n’est pas un procès pratique, et encore moins désirable si l’application n’utilise que quelques méthodes qui ne sont pas dans la librairie.

Donc, les participants de la liste de discussion ont lancé une suggestion : créer un moyen de étendre dynamiquement la librairie. Ce moyen aura dû permettre à l’application de registrer de nouvelles méthodes et les utiliser pendant l’exécution, sans que le code source de la Libwww fût changé avant.

Alors, en profitant les modifications introduites dans la Libwww due au protocole WebDAV, nous avons décidé d’introduire aussi cette idée d’expansion dynamique de la librairie, car cette expansion sera beaucoup utile aux applications qui voudraient utiliser les protocoles liés au WebDAV. Le défi de cette idée était comment introduire ces extensions sans, par autant, changer beaucoup la Libwww. Le principal problème était la structure HTMethod, un *bitset* peu flexible, mais très utilisé dans la librairie.

La structure HTMethod est une énumération, où chaque méthode est identifiée par un seul bit. Ainsi, les méthodes du protocole HTTP occupaient 10 bits et les nouvelles méthodes WebDAV occupaient 7 bits, plus l’indication de méthode invalide, ça faisait 18 possibilités. Internement, le module HTMethod faisait la relation entre ses valeurs et le nom de chaque méthode (une string), en utilisant un vecteur avec les noms possibles.

Pour étendre cette structure sans la modifié gravement, nous avons introduit 7 nouvelles positions dans l’énumération. Ces nouvelles positions ont le but d’identifier sept méthodes dites "d’extension". Ces méthodes ne sont pas définies a priori, elles correspondent à des positions vides dans le vecteur de noms. Cependant, quand une application voudrait utiliser d’autres méthodes qui ne sont pas connues par la Libwww, il lui suffit de registrer sur une de ces méthodes d’extension la nouvelle méthode. Ce registre est fait en utilisant une fonction que nous avons introduite, laquelle remplit la position correspondante du vecteur avec le nom de la nouvelle méthode. Après ce registre, l’application pourrait utiliser cette nouvelle méthode comme si c’était une méthode quelconque, car la Libwww lui reconnaîtra. Par contre, si l’application n’avait pas registré ces méthodes d’extension, toutes les utilisations de ces méthodes sont traitées comme la méthode invalide. Le Cadre 11 présente la définition de ces méthodes d’extension et les fonctions pour leur manipulation.

Les structures :

```
typedef enum {
    METHOD_INVALID      = 0x0,
    ... /* méthodes HTTP */
#ifdef HT_DAV
    ... /* méthodes DAV */
        /* méthodes d'extension */
    METHOD_EXT_0        = 0x20000,
    METHOD_EXT_1        = 0x40000,
    METHOD_EXT_2        = 0x80000,
    METHOD_EXT_3        = 0x100000,
    METHOD_EXT_4        = 0x200000,
    METHOD_EXT_5        = 0x400000,
    METHOD_EXT_6        = 0x800000
    ...
#endif
} HTMethod;
```

```
PRIVATE char *method_names[] =
{
    "INVALID-METHOD",
    ... /* méthodes HTTP */
#ifdef  HT_DAV
    "LOCK",    /* méthodes DAV */
     ...
    "MOVE",
                /* méthodes d'extension */
    NULL,      /* METHOD_EXT_0 */
    NULL,      /* METHOD_EXT_1 */
    NULL,      /* METHOD_EXT_2 */
    NULL,      /* METHOD_EXT_3 */
    NULL,      /* METHOD_EXT_4 */
    NULL,      /* METHOD_EXT_5 */
    NULL,      /* METHOD_EXT_6 */
#endif
```

Les fonctions :

```
#ifdef HT_DAV
extern BOOL HTMethod_setExtensionMethod (HTMethod method, const char * name);
extern BOOL HTMethod_deleteExtensionMethod (HTMethod method);
#endif
```

Cadre 11. *Structure et fonctions des méthodes d'extension de la Libwww. Ceux-ci permettent à l'application d'utiliser des nouvelles méthodes qui ne sont pas, à l'origine, définies dans la librairie. Quand l'application registre une nouvelle méthode, la position correspondante de la structure method_names (au-dessus, à droite) est remplie avec le nom de la nouvelle méthode (pour cela, un indicateur dynamique est utilisé). Quand l'application efface ce registre, cet indicateur est détruit et la position est vidée.*

Néanmoins, il faudrait aussi donner aux applications des moyens pour bien travailler et remplir ses réquisitions qui utilisent les extensions méthodes. Pour cela, nous avons choisi encore une autre suggestion posée à la liste de discussion : l'introduction d'un élément "*message body*" dans l'objet réquisition (`HTRequest`). Cet élément permet aux applications d'introduire dans ses réquisitions des *message body* simples, comme les éléments XML utilisés par les méthodes DAV. Cet élément n'a pas eu d'impact dans le fonctionnement normal de la Libwww, car il n'est utilisé que quand l'application l'a rempli. S'il est vide, il ne sera pas utilisé. En accompagnant ce *message body*, nous avons introduit dans l'objet réquisition des éléments pour indiquer la taille (*message body length*) et le type (*message body format*) de ce *message body*. Une fois l'application a rempli le *message body* et ces éléments dans une réquisition, des *headers* `Content-Length` et `Content-Type` sont gérés et envoyés avec la réquisition.

L'introduction de ce *message body* a impliqué quelques modifications, lesquelles sont détaillées au Cadre 12.

Définition interne (fichier HTReqMan.h) :

```
    #ifdef HT_DAV
        char *              messageBody;
        int                 messageBodyLength;
        HTFormat            messageBodyFormat;
    #endif
```

Fonctions public (fichier HTReq.h) :

```
extern BOOL HTRequest_setMessageBody (HTRequest * request, const char * body);
extern BOOL HTRequest_deleteMessageBody (HTRequest * request);
extern char * HTRequest_messageBody (HTRequest * request);

extern BOOL HTRequest_setMessageBodyLength (HTRequest * request, int length);
extern int HTRequest_messageBodyLength (HTRequest * request);

extern BOOL HTRequest_setMessageBodyFormat (HTRequest * request, HTFormat format);
extern HTFormat HTRequest_messageBodyFormat (HTRequest * request);
```

Modifications internes (fichier HTTPGen.c) :

Fonction "HTTPGenMake", inclusion d'un message body e des *headers* Content−Type et Content−Length si ils sont définis.

Cadre 12. Modifications introduites dans la Libwww pour l'utilisation directe d'un message body simple dans la réquisition.

Toutes ces modifications décrites dans cette section ont été testées par des exemples construits spécialement pour cela. Ces exemples, un pour les méthodes d'extension et autre pour les fonctions DAV, ont été mis ensemble avec d'autres exemples de la Libwww. Ces dernières nous ont permis d'évaluer l'impact que nos modifications ont eu dans les applications qui ne les utilisent pas, et les résultats sont plutôt positifs, car aucune de ces applications semble être dommage.

5 Conclusions

Ce document a décrit l'introduction d'un module pour le protocole WebDAV dans la librairie Libwww. Il a décrit les fonctions du nouveau module et aussi les modifications réalisées dans la librairie. Tout cela permettra l'utilisation du protocole WebDAV dans les applications qui utilisent la Libwww. L'adoption de ce protocole dans un environnement d'édition Web fournit une infrastructure solide pour la création coopérative des pages Web. L'introduction du WebDAV dans la Libwww ira, donc, faciliter la création des modernes applications coopératives avec cette librairie.

Toutes ces modifications ont affecté moins de 5% des fichiers de la Libwww. Et ils ne sont que cinq les modules affectés d'entre tous ces distribués avec la librairie. En dehors ce numéro réduit de fichiers et modules affectés, toutes ces modifications peuvent être ignorées, car elles ont été tachées avec les directives HT_DAV, pour le module HT_DAV et les modifications liées à ceci, et HT_EXT, pour les modifications liées aux méthodes d'extension. Si la directive n'est pas définie en temps de compilation, les modifications correspondantes ne seront pas prises par le compilateur. Et encore, le processus de configuration de la Libwww a été aussi changé pour cela. La configuration standard de la Libwww continue exactement la même, sans aucune référence à ces modifications, ni à les directives HT_DAV et HT_EXT. Toutes ces modifications ne sont qu'activées si la configuration de la Libwww utilise les options "`−−with−dav`" et "−−with−extension". Tout cela

signifie que ceux qui utilisent la Libwww pour développer ses applications ont encore le choix entre utiliser ou non ces modifications.

Quant aux méthodes d'extension de la librairie, ceux−ci représentent une possibilité petite, mais encore efficiente, d'expansion dynamique de la librairie. En utilisant ces méthodes, une application pourra utiliser quelques méthodes normalement inconnues par la Libwww, sans la changer. Cette capacité est encore limitée à sept méthodes, mais si bien reçue par la communauté, nous pourrons l'amplifier jusqu'à quinze méthodes (c'est−à−dire, 32 bits), sans, pour autant, changer beaucoup la structure actuelle.

Finalement, nous avons testé ces modifications avec desj applications conçues pour ça (tous les Cadres utilisés dans la Section 2 de ce document ont été pris avec ces applications de test). Évidemment, autres testes sont bienvenus, car seulement l'utilisation fréquente de ces modifications pourra démontre ces défauts. Pour cela, nous voudrons déposer ces modifications dans la liste de discussion du protocole WebDAV et peut−être les publier dans un site Internet. Nous croyons fortement qui l'utilisation de ce nouveau module DAV pour d'autres personnes, non liés au son développement, sera le moyen le plus efficace de démontre et corriger ses problèmes.

6 Bibliographie

Kahan, J. "Libwww − the W3C Protocol Library". Disponible : http://www.w3.org/Library/. Dernière accès: Fev, 2002.

Kirsch−Pinheiro, M. "Implémentation du WebDAV sur Amaya". Plan du Travail. Disponible : http://gnome.inrialpes.fr:1959/PlanDAV.html. Dernière accèss: Fev, 2002.

Goland, Y.; Whitehead, E.; Faisi, A.; Jensen, D. "HTTP Extensions for Distributed Authoring − WebDAV". RFC 2518. IETF, fev. 99. Disponible : http://andrew2.andrew.cmu.edu/rfc/rfc2518.html. Dernière accès: Sep, 2002.

Vatton, Irène. "Welcome to Amaya". Disponible : http://www.w3.org/Amaya/. Dernière accès : Fev, 2002.

W3C. "www−lib@w3.org Mail Archives". Disponible : http://lists.w3.org/Archives/Public/www−lib/. Dernière accès : Fev, 2002.