Network Working Group   L. Dusseault, Ed.
Request for Comments: 4918   CommerceNet
Obsoletes: 2518   June 2007
Category: Standards Track

      HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)

Abstract

   Web Distributed Authoring and Versioning (WebDAV) consists of a set
   of methods, headers, and content-types ancillary to HTTP/1.1 for the
   management of resource properties, creation and management of
   resource collections, URL namespace manipulation, and resource
   locking (collision avoidance).

   RFC 2518 was published in February 1999, and this specification
   obsoletes RFC 2518 with minor revisions mostly due to
   interoperability experience.

Table of Contents

1. Introduction

   This document describes an extension to the HTTP/1.1 protocol that
   allows clients to perform remote Web content authoring operations.
   This extension provides a coherent set of methods, headers, request
   entity body formats, and response entity body formats that provide
   operations for:

Properties: The ability to create, remove, and query information
about Web pages, such as their authors, creation dates, etc.

Collections: The ability to create sets of documents and to retrieve
a hierarchical membership listing (like a directory listing in a file
system).

Locking: The ability to keep more than one person from working on a
document at the same time. This prevents the "lost update problem",
in which modifications are lost as first one author, then another,
writes changes without merging the other author's changes.

Namespace Operations: The ability to instruct the server to copy and
move Web resources, operations that change the mapping from URLs to
resources.

Requirements and rationale for these operations are described in a
companion document, "Requirements for a Distributed Authoring and
Versioning Protocol for the World Wide Web" [RFC2291].

This document does not specify the versioning operations suggested by
[RFC2291]. That work was done in a separate document, "Versioning
Extensions to WebDAV" [RFC3253].

The sections below provide a detailed introduction to various WebDAV
abstractions: resource properties (Section 4), collections of
resources (Section 5), locks (Section 6) in general, and write locks
(Section 7) specifically.

These abstractions are manipulated by the WebDAV-specific HTTP
methods (Section 9) and the extra HTTP headers (Section 10) used with
WebDAV methods. General considerations for handling HTTP requests
and responses in WebDAV are found in Section 8.

While the status codes provided by HTTP/1.1 are sufficient to
describe most error conditions encountered by WebDAV methods, there
are some errors that do not fall neatly into the existing categories.
This specification defines extra status codes developed for WebDAV
methods (Section 11) and describes existing HTTP status codes
(Section 12) as used in WebDAV. Since some WebDAV methods may

operate over many resources, the Multi-Status response (Section 13)

has been introduced to return status information for multiple resources. Finally, this version of WebDAV introduces precondition and postcondition (Section 16) XML elements in error response bodies.

WebDAV uses XML ([REC-XML]) for property names and some values, and also uses XML to marshal complicated requests and responses. This specification contains DTD and text definitions of all properties (Section 15) and all other XML elements (Section 14) used in marshalling. WebDAV includes a few special rules on extending WebDAV XML marshalling in backwards-compatible ways (Section 17).

Finishing off the specification are sections on what it means for a resource to be compliant with this specification (Section 18), on internationalization support (Section 19), and on security (Section 20).

2. Notational Conventions

Since this document describes a set of extensions to the HTTP/1.1 protocol, the augmented BNF used herein to describe protocol elements is exactly the same as described in Section 2.1 of [RFC2616], including the rules about implied linear whitespace. Since this augmented BNF uses the basic production rules provided in Section 2.2 of [RFC2616], these rules apply to this document as well. Note this is not the standard BNF syntax used in other RFCs.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Note that in natural language, a property like the "creationdate" property in the "DAV:" XML namespace is sometimes referred to as "DAV:creationdate" for brevity.

3. Terminology

URI/URL - A Uniform Resource Identifier and Uniform Resource Locator, respectively. These terms (and the distinction between them) are defined in [RFC3986].

URI/URL Mapping - A relation between an absolute URI and a resource. Since a resource can represent items that are not network retrievable, as well as those that are, it is possible for a resource to have zero, one, or many URI mappings. Mapping a resource to an "http" scheme URI makes it possible to submit HTTP protocol requests to the resource using the URI.

Path Segment - Informally, the characters found between slashes ("/")
in a URI. Formally, as defined in Section 3.3 of [RFC3986].

Collection - Informally, a resource that also acts as a container of
references to child resources. Formally, a resource that contains a
set of mappings between path segments and resources and meets the
requirements defined in Section 5.

Internal Member (of a Collection) - Informally, a child resource of a
collection. Formally, a resource referenced by a path segment
mapping contained in the collection.

Internal Member URL (of a Collection) - A URL of an internal member,
consisting of the URL of the collection (including trailing slash)
plus the path segment identifying the internal member.

Member (of a Collection) - Informally, a "descendant" of a
collection. Formally, an internal member of the collection, or,
recursively, a member of an internal member.

Member URL (of a Collection) - A URL that is either an internal
member URL of the collection itself, or is an internal member URL of
a member of that collection.

Property - A name/value pair that contains descriptive information
about a resource.

Live Property - A property whose semantics and syntax are enforced by
the server. For example, the live property DAV:getcontentlength has
its value, the length of the entity returned by a GET request,
automatically calculated by the server.

Dead Property - A property whose semantics and syntax are not
enforced by the server. The server only records the value of a dead
property; the client is responsible for maintaining the consistency
of the syntax and semantics of a dead property.

Principal - A distinct human or computational actor that initiates
access to network resources.

State Token - A URI that represents a state of a resource. Lock
tokens are the only state tokens defined in this specification.

[4](#). Data Model for Resource Properties

[4.1](#). The Resource Property Model

   Properties are pieces of data that describe the state of a resource.
   Properties are data about data.

   Properties are used in distributed authoring environments to provide
   for efficient discovery and management of resources. For example, a
   'subject' property might allow for the indexing of all resources by
   their subject, and an 'author' property might allow for the discovery
   of what authors have written which documents.

   The DAV property model consists of name/value pairs. The name of a
   property identifies the property's syntax and semantics, and provides
   an address by which to refer to its syntax and semantics.

   There are two categories of properties: "live" and "dead". A live
   property has its syntax and semantics enforced by the server. Live
   properties include cases where a) the value of a property is
   protected and maintained by the server, and b) the value of the
   property is maintained by the client, but the server performs syntax
   checking on submitted values. All instances of a given live property
   MUST comply with the definition associated with that property name.
   A dead property has its syntax and semantics enforced by the client;
   the server merely records the value of the property verbatim.

[4.2](#). Properties and HTTP Headers

   Properties already exist, in a limited sense, in HTTP message
   headers. However, in distributed authoring environments, a
   relatively large number of properties are needed to describe the
   state of a resource, and setting/returning them all through HTTP
   headers is inefficient. Thus, a mechanism is needed that allows a
   principal to identify a set of properties in which the principal is
   interested and to set or retrieve just those properties.

[4.3](#). Property Values

The value of a property is always a (well-formed) XML fragment.

XML has been chosen because it is a flexible, self-describing, structured data format that supports rich schema definitions, and because of its support for multiple character sets. XML's self-describing nature allows any property's value to be extended by adding elements. Clients will not break when they encounter extensions because they will still have the data specified in the original schema and MUST ignore elements they do not understand.

XML's support for multiple character sets allows any human-readable property to be encoded and read in a character set familiar to the user. XML's support for multiple human languages, using the "xml:lang" attribute, handles cases where the same character set is employed by multiple human languages. Note that xml:lang scope is recursive, so an xml:lang attribute on any element containing a property name element applies to the property value unless it has been overridden by a more locally scoped attribute. Note that a property only has one value, in one language (or language MAY be left undefined); a property does not have multiple values in different languages or a single value in multiple languages.

A property is always represented with an XML element consisting of the property name, called the "property name element". The simplest example is an empty property, which is different from a property that does not exist:

    <R:title xmlns:R="http://www.example.com/ns/"></R:title>

The value of the property appears inside the property name element. The value may be any kind of well-formed XML content, including both text-only and mixed content. Servers MUST preserve the following XML Information Items (using the terminology from [REC-XML-INFOSET]) in storage and transmission of dead properties:

For the property name Element Information Item itself:

    [namespace name]

    [local name]

    [attributes] named "xml:lang" or any such attribute in scope

[children] of type element or character

On all Element Information Items in the property value:

[namespace name]

[local name]

[attributes]

[children] of type element or character

On Attribute Information Items in the property value:

[namespace name]

[local name]

[normalized value]

On Character Information Items in the property value:

[character code]

Since prefixes are used in some XML vocabularies (XPath and XML Schema, for example), servers SHOULD preserve, for any Information Item in the value:

[prefix]

XML Infoset attributes not listed above MAY be preserved by the server, but clients MUST NOT rely on them being preserved. The above rules would also apply by default to live properties, unless defined otherwise.

Servers MUST ignore the XML attribute xml:space if present and never use it to change whitespace handling. Whitespace in property values

is significant.

4.3.1. Example - Property with Mixed Content

Consider a dead property 'author' created by the client as follows:

```
<D:prop xml:lang="en" xmlns:D="DAV:">
  <x:author xmlns:x='http://example.com/ns'>
    <x:name>Jane Doe</x:name>
    <!-- Jane's contact info -->
    <x:uri type='email'
           added='2005-11-26'>mailto:jane.doe@example.com</x:uri>
    <x:uri type='web'
           added='2005-11-27'>http://www.example.com</x:uri>
    <x:notes xmlns:h='http://www.w3.org/1999/xhtml'>
      Jane has been working way <h:em>too</h:em> long on the
      long-awaited revision of <![CDATA[<RFC2518>]]>.
    </x:notes>
  </x:author>
</D:prop>
```

When this property is requested, a server might return:

```
<D:prop xmlns:D='DAV:'><author
        xml:lang='en'
        xmlns:x='http://example.com/ns'
        xmlns='http://example.com/ns'
        xmlns:h='http://www.w3.org/1999/xhtml'>
    <x:name>Jane Doe</x:name>
    <x:uri   added="2005-11-26" type="email"
      >mailto:jane.doe@example.com</x:uri>
    <x:uri   added="2005-11-27" type="web"
      >http://www.example.com</x:uri>
    <x:notes>
      Jane has been working way <h:em>too</h:em> long on the
      long-awaited revision of &lt;RFC2518&gt;.
    </x:notes>
  </author>
</D:prop>
```

Note in this example:

o   The [prefix] for the property name itself was not preserved, being
    non-significant, whereas all other [prefix] values have been
    preserved,

o   attribute values have been rewritten with double quotes instead of
    single quotes (quoting style is not significant), and attribute
    order has not been preserved,

o   the xml:lang attribute has been returned on the property name
    element itself (it was in scope when the property was set, but the
    exact position in the response is not considered significant as
    long as it is in scope),

o   whitespace between tags has been preserved everywhere (whitespace
    between attributes not so),

o   CDATA encapsulation was replaced with character escaping (the
    reverse would also be legal),

o   the comment item was stripped (as would have been a processing
    instruction item).

Implementation note: there are cases such as editing scenarios where
clients may require that XML content is preserved character by
character (such as attribute ordering or quoting style). In this
case, clients should consider using a text-only property value by
escaping all characters that have a special meaning in XML parsing.

4.4. Property Names

   A property name is a universally unique identifier that is associated
   with a schema that provides information about the syntax and
   semantics of the property.

   Because a property's name is universally unique, clients can depend
   upon consistent behavior for a particular property across multiple
   resources, on the same and across different servers, so long as that
   property is "live" on the resources in question, and the
   implementation of the live property is faithful to its definition.

The XML namespace mechanism, which is based on URIs ([RFC3986]), is used to name properties because it prevents namespace collisions and provides for varying degrees of administrative control.

The property namespace is flat; that is, no hierarchy of properties is explicitly recognized. Thus, if a property A and a property A/B exist on a resource, there is no recognition of any relationship between the two properties. It is expected that a separate specification will eventually be produced that will address issues relating to hierarchical properties.

Finally, it is not possible to define the same property twice on a single resource, as this would cause a collision in the resource's property namespace.

4.5. Source Resources and Output Resources

Some HTTP resources are dynamically generated by the server. For these resources, there presumably exists source code somewhere governing how that resource is generated. The relationship of source files to output HTTP resources may be one to one, one to many, many to one, or many to many. There is no mechanism in HTTP to determine whether a resource is even dynamic, let alone where its source files exist or how to author them. Although this problem would usefully be solved, interoperable WebDAV implementations have been widely deployed without actually solving this problem, by dealing only with static resources. Thus, the source vs. output problem is not solved in this specification and has been deferred to a separate document.

5. Collections of Web Resources

This section provides a description of a type of Web resource, the collection, and discusses its interactions with the HTTP URL namespace and with HTTP methods. The purpose of a collection resource is to model collection-like objects (e.g., file system directories) within a server's namespace.

All DAV-compliant resources MUST support the HTTP URL namespace model specified herein.

5.1. HTTP URL Namespace Model

The HTTP URL namespace is a hierarchical namespace where the
hierarchy is delimited with the "/" character.

An HTTP URL namespace is said to be consistent if it meets the
following conditions: for every URL in the HTTP hierarchy there
exists a collection that contains that URL as an internal member URL.
The root, or top-level collection of the namespace under
consideration, is exempt from the previous rule. The top-level
collection of the namespace under consideration is not necessarily
the collection identified by the absolute path '/' -- it may be
identified by one or more path segments (e.g., /servlets/webdav/...)

Neither HTTP/1.1 nor WebDAV requires that the entire HTTP URL
namespace be consistent -- a WebDAV-compatible resource may not have
a parent collection. However, certain WebDAV methods are prohibited
from producing results that cause namespace inconsistencies.

As is implicit in [RFC2616] and [RFC3986], any resource, including
collection resources, MAY be identified by more than one URI. For
example, a resource could be identified by multiple HTTP URLs.

5.2. Collection Resources

Collection resources differ from other resources in that they also
act as containers. Some HTTP methods apply only to a collection, but
some apply to some or all of the resources inside the container
defined by the collection. When the scope of a method is not clear,
the client can specify what depth to apply. Depth can be either zero
levels (only the collection), one level (the collection and directly
contained resources), or infinite levels (the collection and all
contained resources recursively).

A collection's state consists of at least a set of mappings between
path segments and resources, and a set of properties on the
collection itself. In this document, a resource B will be said to be
contained in the collection resource A if there is a path segment
mapping that maps to B and that is contained in A. A collection MUST
contain at most one mapping for a given path segment, i.e., it is
illegal to have the same path segment mapped to more than one
resource.

Properties defined on collections behave exactly as do properties on
non-collection resources. A collection MAY have additional state
such as entity bodies returned by GET.

For all WebDAV-compliant resources A and B, identified by URLs "U"
and "V", respectively, such that "V" is equal to "U/SEGMENT", A MUST
be a collection that contains a mapping from "SEGMENT" to B. So, if
resource B with URL "http://example.com/bar/blah" is WebDAV compliant
and if resource A with URL "http://example.com/bar/" is WebDAV
compliant, then resource A must be a collection and must contain
exactly one mapping from "blah" to B.

Although commonly a mapping consists of a single segment and a
resource, in general, a mapping consists of a set of segments and a
resource. This allows a server to treat a set of segments as
equivalent (i.e., either all of the segments are mapped to the same
resource, or none of the segments are mapped to a resource). For
example, a server that performs case-folding on segments will treat
the segments "ab", "Ab", "aB", and "AB" as equivalent. A client can
then use any of these segments to identify the resource. Note that a
PROPFIND result will select one of these equivalent segments to
identify the mapping, so there will be one PROPFIND response element
per mapping, not one per segment in the mapping.

Collection resources MAY have mappings to non-WebDAV-compliant
resources in the HTTP URL namespace hierarchy but are not required to
do so. For example, if resource X with URL
"http://example.com/bar/blah" is not WebDAV compliant and resource A
with "URL http://example.com/bar/" identifies a WebDAV collection,
then A may or may not have a mapping from "blah" to X.

If a WebDAV-compliant resource has no WebDAV-compliant internal
members in the HTTP URL namespace hierarchy, then the WebDAV-
compliant resource is not required to be a collection.

There is a standing convention that when a collection is referred to
by its name without a trailing slash, the server MAY handle the
request as if the trailing slash were present. In this case, it
SHOULD return a Content-Location header in the response, pointing to
the URL ending with the "/". For example, if a client invokes a
method on http://example.com/blah (no trailing slash), the server may
respond as if the operation were invoked on http://example.com/blah/
(trailing slash), and should return a Content-Location header with
the value http://example.com/blah/. Wherever a server produces a URL
referring to a collection, the server SHOULD include the trailing
slash. In general, clients SHOULD use the trailing slash form of
collection names. If clients do not use the trailing slash form the

client needs to be prepared to see a redirect response. Clients will

find the DAV:resourcetype property more reliable than the URL to find
out if a resource is a collection.

Clients MUST be able to support the case where WebDAV resources are
contained inside non-WebDAV resources. For example, if an OPTIONS
response from "http://example.com/servlet/dav/collection" indicates
WebDAV support, the client cannot assume that
"http://example.com/servlet/dav/" or its parent necessarily are
WebDAV collections.

A typical scenario in which mapped URLs do not appear as members of
their parent collection is the case where a server allows links or
redirects to non-WebDAV resources. For instance, "/col/link" might
not appear as a member of "/col/", although the server would respond
with a 302 status to a GET request to "/col/link"; thus, the URL
"/col/link" would indeed be mapped. Similarly, a dynamically-
generated page might have a URL mapping from "/col/index.html", thus
this resource might respond with a 200 OK to a GET request yet not
appear as a member of "/col/".

Some mappings to even WebDAV-compliant resources might not appear in
the parent collection. An example for this case are servers that
support multiple alias URLs for each WebDAV-compliant resource. A
server may implement case-insensitive URLs, thus "/col/a" and
"/col/A" identify the same resource, yet only either "a" or "A" is
reported upon listing the members of "/col". In cases where a server
treats a set of segments as equivalent, the server MUST expose only
one preferred segment per mapping, consistently chosen, in PROPFIND
responses.

6. Locking

The ability to lock a resource provides a mechanism for serializing
access to that resource. Using a lock, an authoring client can
provide a reasonable guarantee that another principal will not modify
a resource while it is being edited. In this way, a client can
prevent the "lost update" problem.

This specification allows locks to vary over two client-specified
parameters, the number of principals involved (exclusive vs. shared)

and the type of access to be granted. This document defines locking
for only one access type, write. However, the syntax is extensible,
and permits the eventual specification of locking for other access
types.

6.1. Lock Model

   This section provides a concise model for how locking behaves. Later
   sections will provide more detail on some of the concepts and refer
   back to these model statements. Normative statements related to LOCK
   and UNLOCK method handling can be found in the sections on those
   methods, whereas normative statements that cover any method are
   gathered here.

   1. A lock either directly or indirectly locks a resource.

   2. A resource becomes directly locked when a LOCK request to a URL
       of that resource creates a new lock. The "lock-root" of the new
       lock is that URL. If at the time of the request, the URL is not
       mapped to a resource, a new empty resource is created and
       directly locked.

   3. An exclusive lock (Section 6.2) conflicts with any other kind of
       lock on the same resource, whether either lock is direct or
       indirect. A server MUST NOT create conflicting locks on a
       resource.

   4. For a collection that is locked with a depth-infinity lock L, all
       member resources are indirectly locked. Changes in membership of
       such a collection affect the set of indirectly locked resources:

       * If a member resource is added to the collection, the new
          member resource MUST NOT already have a conflicting lock,
          because the new resource MUST become indirectly locked by L.

       * If a member resource stops being a member of the collection,
          then the resource MUST no longer be indirectly locked by L.

5. Each lock is identified by a single globally unique lock token
    ([Section 6.5](#)).

6. An UNLOCK request deletes the lock with the specified lock token.
    After a lock is deleted, no resource is locked by that lock.

7. A lock token is "submitted" in a request when it appears in an
    "If" header ([Section 7](#), "Write Lock", discusses when token
    submission is required for write locks).

8. If a request causes the lock-root of any lock to become an
    unmapped URL, then the lock MUST also be deleted by that request.

6.2. Exclusive vs. Shared Locks

   The most basic form of lock is an exclusive lock. Exclusive locks
   avoid having to deal with content change conflicts, without requiring
   any coordination other than the methods described in this
   specification.

   However, there are times when the goal of a lock is not to exclude
   others from exercising an access right but rather to provide a
   mechanism for principals to indicate that they intend to exercise
   their access rights. Shared locks are provided for this case. A
   shared lock allows multiple principals to receive a lock. Hence any
   principal that has both access privileges and a valid lock can use
   the locked resource.

   With shared locks, there are two trust sets that affect a resource.
   The first trust set is created by access permissions. Principals who
   are trusted, for example, may have permission to write to the
   resource. Among those who have access permission to write to the
   resource, the set of principals who have taken out a shared lock also
   must trust each other, creating a (typically) smaller trust set
   within the access permission write set.

   Starting with every possible principal on the Internet, in most
   situations the vast majority of these principals will not have write
   access to a given resource. Of the small number who do have write

access, some principals may decide to guarantee their edits are free
from overwrite conflicts by using exclusive write locks. Others may
decide they trust their collaborators will not overwrite their work
(the potential set of collaborators being the set of principals who
have write permission) and use a shared lock, which informs their
collaborators that a principal may be working on the resource.

The WebDAV extensions to HTTP do not need to provide all of the
communications paths necessary for principals to coordinate their
activities. When using shared locks, principals may use any out-of-
band communication channel to coordinate their work (e.g., face-to-
face interaction, written notes, post-it notes on the screen,
telephone conversation, email, etc.) The intent of a shared lock is
to let collaborators know who else may be working on a resource.

Shared locks are included because experience from Web-distributed
authoring systems has indicated that exclusive locks are often too
rigid. An exclusive lock is used to enforce a particular editing
process: take out an exclusive lock, read the resource, perform
edits, write the resource, release the lock. This editing process
has the problem that locks are not always properly released, for
example, when a program crashes or when a lock creator leaves without

unlocking a resource. While both timeouts ([Section 6.6](#)) and
administrative action can be used to remove an offending lock,
neither mechanism may be available when needed; the timeout may be
long or the administrator may not be available.

A successful request for a new shared lock MUST result in the
generation of a unique lock associated with the requesting principal.
Thus, if five principals have taken out shared write locks on the
same resource, there will be five locks and five lock tokens, one for
each principal.

[6.3](#). Required Support

A WebDAV-compliant resource is not required to support locking in any
form. If the resource does support locking, it may choose to support
any combination of exclusive and shared locks for any access types.

The reason for this flexibility is that locking policy strikes to the
very heart of the resource management and versioning systems employed

by various storage repositories. These repositories require control
over what sort of locking will be made available. For example, some
repositories only support shared write locks, while others only
provide support for exclusive write locks, while yet others use no
locking at all. As each system is sufficiently different to merit
exclusion of certain locking features, this specification leaves
locking as the sole axis of negotiation within WebDAV.

6.4. Lock Creator and Privileges

The creator of a lock has special privileges to use the lock to
modify the resource. When a locked resource is modified, a server
MUST check that the authenticated principal matches the lock creator
(in addition to checking for valid lock token submission).

The server MAY allow privileged users other than the lock creator to
destroy a lock (for example, the resource owner or an administrator).
The 'unlock' privilege in [RFC3744] was defined to provide that
permission.

There is no requirement for servers to accept LOCK requests from all
users or from anonymous users.

Note that having a lock does not confer full privilege to modify the
locked resource. Write access and other privileges MUST be enforced
through normal privilege or authentication mechanisms, not based on
the possible obscurity of lock token values.

6.5. Lock Tokens

A lock token is a type of state token that identifies a particular
lock. Each lock has exactly one unique lock token generated by the
server. Clients MUST NOT attempt to interpret lock tokens in any
way.

Lock token URIs MUST be unique across all resources for all time.
This uniqueness constraint allows lock tokens to be submitted across
resources and servers without fear of confusion. Since lock tokens
are unique, a client MAY submit a lock token in an If header on a
resource other than the one that returned it.

When a LOCK operation creates a new lock, the new lock token is
returned in the Lock-Token response header defined in Section 10.5,
and also in the body of the response.

Servers MAY make lock tokens publicly readable (e.g., in the DAV:
lockdiscovery property). One use case for making lock tokens
readable is so that a long-lived lock can be removed by the resource
owner (the client that obtained the lock might have crashed or
disconnected before cleaning up the lock). Except for the case of
using UNLOCK under user guidance, a client SHOULD NOT use a lock
token created by another client instance.

This specification encourages servers to create Universally Unique
Identifiers (UUIDs) for lock tokens, and to use the URI form defined
by "A Universally Unique Identifier (UUID) URN Namespace"
([RFC4122]). However, servers are free to use any URI (e.g., from
another scheme) so long as it meets the uniqueness requirements. For
example, a valid lock token might be constructed using the
"opaquelocktoken" scheme defined in Appendix C.

Example: "urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6"

6.6. Lock Timeout

A lock MAY have a limited lifetime. The lifetime is suggested by the
client when creating or refreshing the lock, but the server
ultimately chooses the timeout value. Timeout is measured in seconds
remaining until lock expiration.

The timeout counter MUST be restarted if a refresh lock request is
successful (see Section 9.10.2). The timeout counter SHOULD NOT be
restarted at any other time.

If the timeout expires, then the lock SHOULD be removed. In this
case the server SHOULD act as if an UNLOCK method was executed by the

server on the resource using the lock token of the timed-out lock,
performed with its override authority.

Servers are advised to pay close attention to the values submitted by
clients, as they will be indicative of the type of activity the

client intends to perform. For example, an applet running in a
browser may need to lock a resource, but because of the instability
of the environment within which the applet is running, the applet may
be turned off without warning. As a result, the applet is likely to
ask for a relatively small timeout value so that if the applet dies,
the lock can be quickly harvested. However, a document management
system is likely to ask for an extremely long timeout because its
user may be planning on going offline.

A client MUST NOT assume that just because the timeout has expired,
the lock has immediately been removed.

Likewise, a client MUST NOT assume that just because the timeout has
not expired, the lock still exists. Clients MUST assume that locks
can arbitrarily disappear at any time, regardless of the value given
in the Timeout header. The Timeout header only indicates the
behavior of the server if extraordinary circumstances do not occur.
For example, a sufficiently privileged user may remove a lock at any
time, or the system may crash in such a way that it loses the record
of the lock's existence.

6.7. Lock Capability Discovery

Since server lock support is optional, a client trying to lock a
resource on a server can either try the lock and hope for the best,
or perform some form of discovery to determine what lock capabilities
the server supports. This is known as lock capability discovery. A
client can determine what lock types the server supports by
retrieving the DAV:supportedlock property.

Any DAV-compliant resource that supports the LOCK method MUST support
the DAV:supportedlock property.

6.8. Active Lock Discovery

If another principal locks a resource that a principal wishes to
access, it is useful for the second principal to be able to find out
who the first principal is. For this purpose the DAV:lockdiscovery
property is provided. This property lists all outstanding locks,
describes their type, and MAY even provide the lock tokens.

Any DAV-compliant resource that supports the LOCK method MUST support
the DAV:lockdiscovery property.

[7](#). Write Lock

   This section describes the semantics specific to the write lock type.
   The write lock is a specific instance of a lock type, and is the only
   lock type described in this specification.

   An exclusive write lock protects a resource: it prevents changes by
   any principal other than the lock creator and in any case where the
   lock token is not submitted (e.g., by a client process other than the
   one holding the lock).

   Clients MUST submit a lock-token they are authorized to use in any
   request that modifies a write-locked resource. The list of
   modifications covered by a write-lock include:

   1. A change to any of the following aspects of any write-locked
       resource:

       * any variant,

       * any dead property,

       * any live property that is lockable (a live property is
          lockable unless otherwise defined.)

   2. For collections, any modification of an internal member URI. An
       internal member URI of a collection is considered to be modified
       if it is added, removed, or identifies a different resource.
       More discussion on write locks and collections is found in
       [Section 7.4](#).

   3. A modification of the mapping of the root of the write lock,
       either to another resource or to no resource (e.g., DELETE).

   Of the methods defined in HTTP and WebDAV, PUT, POST, PROPPATCH,
   LOCK, UNLOCK, MOVE, COPY (for the destination resource), DELETE, and
   MKCOL are affected by write locks. All other HTTP/WebDAV methods
   defined so far -- GET in particular -- function independently of a
   write lock.

   The next few sections describe in more specific terms how write locks
   interact with various operations.

[7.1](#). Write Locks and Properties

   While those without a write lock may not alter a property on a
   resource it is still possible for the values of live properties to
   change, even while locked, due to the requirements of their schemas.
   Only dead properties and live properties defined as lockable are
   guaranteed not to change while write locked.

[7.2](#). Avoiding Lost Updates

   Although the write locks provide some help in preventing lost
   updates, they cannot guarantee that updates will never be lost.
   Consider the following scenario:

   Two clients A and B are interested in editing the resource
   'index.html'. Client A is an HTTP client rather than a WebDAV
   client, and so does not know how to perform locking.

   Client A doesn't lock the document, but does a GET, and begins
   editing.

   Client B does LOCK, performs a GET and begins editing.

   Client B finishes editing, performs a PUT, then an UNLOCK.

   Client A performs a PUT, overwriting and losing all of B's changes.

   There are several reasons why the WebDAV protocol itself cannot
   prevent this situation. First, it cannot force all clients to use
   locking because it must be compatible with HTTP clients that do not
   comprehend locking. Second, it cannot require servers to support
   locking because of the variety of repository implementations, some of
   which rely on reservations and merging rather than on locking.
   Finally, being stateless, it cannot enforce a sequence of operations
   like LOCK / GET / PUT / UNLOCK.

   WebDAV servers that support locking can reduce the likelihood that
   clients will accidentally overwrite each other's changes by requiring
   clients to lock resources before modifying them. Such servers would
   effectively prevent HTTP 1.0 and HTTP 1.1 clients from modifying
   resources.

WebDAV clients can be good citizens by using a lock / retrieve /
write /unlock sequence of operations (at least by default) whenever
they interact with a WebDAV server that supports locking.

HTTP 1.1 clients can be good citizens, avoiding overwriting other
clients' changes, by using entity tags in If-Match headers with any
requests that would modify resources.

Information managers may attempt to prevent overwrites by
implementing client-side procedures requiring locking before
modifying WebDAV resources.

7.3. Write Locks and Unmapped URLs

WebDAV provides the ability to send a LOCK request to an unmapped URL
in order to reserve the name for use. This is a simple way to avoid
the lost-update problem on the creation of a new resource (another
way is to use If-None-Match header specified in Section 14.26 of
[RFC2616]). It has the side benefit of locking the new resource
immediately for use of the creator.

Note that the lost-update problem is not an issue for collections
because MKCOL can only be used to create a collection, not to
overwrite an existing collection. When trying to lock a collection
upon creation, clients can attempt to increase the likelihood of
getting the lock by pipelining the MKCOL and LOCK requests together
(but because this doesn't convert two separate operations into one
atomic operation, there's no guarantee this will work).

A successful lock request to an unmapped URL MUST result in the
creation of a locked (non-collection) resource with empty content.
Subsequently, a successful PUT request (with the correct lock token)
provides the content for the resource. Note that the LOCK request
has no mechanism for the client to provide Content-Type or Content-
Language, thus the server will use defaults or empty values and rely
on the subsequent PUT request for correct values.

A resource created with a LOCK is empty but otherwise behaves in
every way as a normal resource. It behaves the same way as a

resource created by a PUT request with an empty body (and where a
Content-Type and Content-Language was not specified), followed by a
LOCK request to the same resource. Following from this model, a
locked empty resource:

o Can be read, deleted, moved, and copied, and in all ways behaves
   as a regular non-collection resource.

o Appears as a member of its parent collection.

o SHOULD NOT disappear when its lock goes away (clients must
   therefore be responsible for cleaning up their own mess, as with
   any other operation or any non-empty resource).

o MAY NOT have values for properties like DAV:getcontentlanguage
   that haven't been specified yet by the client.

o Can be updated (have content added) with a PUT request.

o MUST NOT be converted into a collection. The server MUST fail a
   MKCOL request (as it would with a MKCOL request to any existing
   non-collection resource).

o MUST have defined values for DAV:lockdiscovery and DAV:
   supportedlock properties.

o The response MUST indicate that a resource was created, by use of
   the "201 Created" response code (a LOCK request to an existing
   resource instead will result in 200 OK). The body must still
   include the DAV:lockdiscovery property, as with a LOCK request to
   an existing resource.

The client is expected to update the locked empty resource shortly
after locking it, using PUT and possibly PROPPATCH.

Alternatively and for backwards compatibility to [[RFC2518](#)], servers
MAY implement Lock-Null Resources (LNRs) instead (see definition in
[Appendix D](#)). Clients can easily interoperate both with servers that
support the old model LNRs and the recommended model of "locked empty
resources" by only attempting PUT after a LOCK to an unmapped URL,
not MKCOL or GET, and by not relying on specific properties of LNRs.

7.4. Write Locks and Collections

   There are two kinds of collection write locks. A depth-0 write lock
   on a collection protects the collection properties plus the internal
   member URLs of that one collection, while not protecting the content
   or properties of member resources (if the collection itself has any
   entity bodies, those are also protected). A depth-infinity write
   lock on a collection provides the same protection on that collection
   and also provides write lock protection on every member resource.

   Expressed otherwise, a write lock of either kind protects any request
   that would create a new resource in a write locked collection, any
   request that would remove an internal member URL of a write locked
   collection, and any request that would change the segment name of any
   internal member.

   Thus, a collection write lock protects all the following actions:

   o DELETE a collection's direct internal member,

   o MOVE an internal member out of the collection,

   o MOVE an internal member into the collection,

   o MOVE to rename an internal member within a collection,

   o COPY an internal member into a collection, and

   o PUT or MKCOL request that would create a new internal member.

   The collection's lock token is required in addition to the lock token
   on the internal member itself, if it is locked separately.

   In addition, a depth-infinity lock affects all write operations to
   all members of the locked collection. With a depth-infinity lock,
   the resource identified by the root of the lock is directly locked,
   and all its members are indirectly locked.

   o Any new resource added as a descendant of a depth-infinity locked
      collection becomes indirectly locked.

o Any indirectly locked resource moved out of the locked collection
      into an unlocked collection is thereafter unlocked.

   o Any indirectly locked resource moved out of a locked source
      collection into a depth-infinity locked target collection remains
      indirectly locked but is now protected by the lock on the target
      collection (the target collection's lock token will thereafter be
      required to make further changes).

   If a depth-infinity write LOCK request is issued to a collection
   containing member URLs identifying resources that are currently
   locked in a manner that conflicts with the new lock (see Section 6.1,
   point 3), the request MUST fail with a 423 (Locked) status code, and
   the response SHOULD contain the 'no-conflicting-lock' precondition.

   If a lock request causes the URL of a resource to be added as an
   internal member URL of a depth-infinity locked collection, then the
   new resource MUST be automatically protected by the lock. For
   example, if the collection /a/b/ is write locked and the resource /c
   is moved to /a/b/c, then resource /a/b/c will be added to the write
   lock.

7.5. Write Locks and the If Request Header

   A user agent has to demonstrate knowledge of a lock when requesting
   an operation on a locked resource. Otherwise, the following scenario
   might occur. In the scenario, program A, run by User A, takes out a
   write lock on a resource. Program B, also run by User A, has no
   knowledge of the lock taken out by program A, yet performs a PUT to
   the locked resource. In this scenario, the PUT succeeds because
   locks are associated with a principal, not a program, and thus
   program B, because it is acting with principal A's credential, is
   allowed to perform the PUT. However, had program B known about the
   lock, it would not have overwritten the resource, preferring instead
   to present a dialog box describing the conflict to the user. Due to
   this scenario, a mechanism is needed to prevent different programs

from accidentally ignoring locks taken out by other programs with the
same authorization.

In order to prevent these collisions, a lock token MUST be submitted
by an authorized principal for all locked resources that a method may
change or the method MUST fail. A lock token is submitted when it
appears in an If header. For example, if a resource is to be moved
and both the source and destination are locked, then two lock tokens
must be submitted in the If header, one for the source and the other
for the destination.

7.5.1. Example - Write Lock and COPY

   >>Request

     COPY /~fielding/index.html HTTP/1.1
     Host: www.example.com
     Destination: http://www.example.com/users/f/fielding/index.html
     If: <http://www.example.com/users/f/fielding/index.html>
         (<urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6>)

   >>Response

     HTTP/1.1 204 No Content

In this example, even though both the source and destination are
locked, only one lock token must be submitted (the one for the lock
on the destination). This is because the source resource is not
modified by a COPY, and hence unaffected by the write lock. In this
example, user agent authentication has previously occurred via a
mechanism outside the scope of the HTTP protocol, in the underlying
transport layer.

7.5.2. Example - Deleting a Member of a Locked Collection

Consider a collection "/locked" with an exclusive, depth-infinity
write lock, and an attempt to delete an internal member "/locked/
member":

   >>Request

```
      DELETE /locked/member HTTP/1.1
      Host: example.com

   >>Response

      HTTP/1.1 423 Locked
      Content-Type: application/xml; charset="utf-8"
      Content-Length: xxxx

      <?xml version="1.0" encoding="utf-8" ?>
      <D:error xmlns:D="DAV:">
        <D:lock-token-submitted>
          <D:href>/locked/</D:href>
        </D:lock-token-submitted>
      </D:error>
```

Thus, the client would need to submit the lock token with the request
to make it succeed. To do that, various forms of the If header (see
[Section 10.4](#)) could be used.

"No-Tag-List" format:

```
   If: (<urn:uuid:150852e2-3847-42d5-8cbe-0f4f296f26cf>)
```

"Tagged-List" format, for "http://example.com/locked/":

```
   If: <http://example.com/locked/>
       (<urn:uuid:150852e2-3847-42d5-8cbe-0f4f296f26cf>)
```

"Tagged-List" format, for "http://example.com/locked/member":

```
   If: <http://example.com/locked/member>
       (<urn:uuid:150852e2-3847-42d5-8cbe-0f4f296f26cf>)
```

Note that, for the purpose of submitting the lock token, the actual
form doesn't matter; what's relevant is that the lock token appears
in the If header, and that the If header itself evaluates to true.

. Write Locks and COPY/MOVE

   A COPY method invocation MUST NOT duplicate any write locks active on
   the source. However, as previously noted, if the COPY copies the
   resource into a collection that is locked with a depth-infinity lock,
   then the resource will be added to the lock.

   A successful MOVE request on a write locked resource MUST NOT move
   the write lock with the resource. However, if there is an existing
   lock at the destination, the server MUST add the moved resource to
   the destination lock scope. For example, if the MOVE makes the
   resource a child of a collection that has a depth-infinity lock, then
   the resource will be added to that collection's lock. Additionally,
   if a resource with a depth-infinity lock is moved to a destination
   that is within the scope of the same lock (e.g., within the URL
   namespace tree covered by the lock), the moved resource will again be
   added to the lock. In both these examples, as specified in
   , an If header must be submitted containing a lock token
   for both the source and destination.

. Refreshing Write Locks

   A client MUST NOT submit the same write lock request twice. Note
   that a client is always aware it is resubmitting the same lock
   request because it must include the lock token in the If header in
   order to make the request for a resource that is already locked.

   However, a client may submit a LOCK request with an If header but
   without a body. A server receiving a LOCK request with no body MUST
   NOT create a new lock -- this form of the LOCK request is only to be
   used to "refresh" an existing lock (meaning, at minimum, that any
   timers associated with the lock MUST be reset).

   Clients may submit Timeout headers of arbitrary value with their lock
   refresh requests. Servers, as always, may ignore Timeout headers
   submitted by the client, and a server MAY refresh a lock with a
   timeout period that is different than the previous timeout period
   used for the lock, provided it advertises the new value in the LOCK
   refresh response.

   If an error is received in response to a refresh LOCK request, the
   client MUST NOT assume that the lock was refreshed.

8. General Request and Response Handling

8.1. Precedence in Error Handling

   Servers MUST return authorization errors in preference to other
   errors. This avoids leaking information about protected resources
   (e.g., a client that finds that a hidden resource exists by seeing a
   423 Locked response to an anonymous request to the resource).

8.2. Use of XML

   In HTTP/1.1, method parameter information was exclusively encoded in
   HTTP headers. Unlike HTTP/1.1, WebDAV encodes method parameter
   information either in an XML ([REC-XML]) request entity body, or in
   an HTTP header. The use of XML to encode method parameters was
   motivated by the ability to add extra XML elements to existing
   structures, providing extensibility; and by XML's ability to encode
   information in ISO 10646 character sets, providing
   internationalization support.

   In addition to encoding method parameters, XML is used in WebDAV to
   encode the responses from methods, providing the extensibility and
   internationalization advantages of XML for method output, as well as
   input.

   When XML is used for a request or response body, the Content-Type
   type SHOULD be application/xml. Implementations MUST accept both
   text/xml and application/xml in request and response bodies. Use of
   text/xml is deprecated.

   All DAV-compliant clients and resources MUST use XML parsers that are
   compliant with [REC-XML] and [REC-XML-NAMES]. All XML used in either
   requests or responses MUST be, at minimum, well formed and use
   namespaces correctly. If a server receives XML that is not well-
   formed, then the server MUST reject the entire request with a 400
   (Bad Request). If a client receives XML that is not well-formed in a
   response, then the client MUST NOT assume anything about the outcome
   of the executed method and SHOULD treat the server as malfunctioning.

   Note that processing XML submitted by an untrusted source may cause
   risks connected to privacy, security, and service quality (see
   [Section 20](#)). Servers MAY reject questionable requests (even though
   they consist of well-formed XML), for instance, with a 400 (Bad

Request) status code and an optional response body explaining the
problem.

8.3. URL Handling

   URLs appear in many places in requests and responses.
   Interoperability experience with [RFC2518] showed that many clients
   parsing Multi-Status responses did not fully implement the full
   Reference Resolution defined in Section 5 of [RFC3986]. Thus,
   servers in particular need to be careful in handling URLs in
   responses, to ensure that clients have enough context to be able to
   interpret all the URLs. The rules in this section apply not only to
   resource URLs in the 'href' element in Multi-Status responses, but
   also to the Destination and If header resource URLs.

   The sender has a choice between two approaches: using a relative
   reference, which is resolved against the Request-URI, or a full URI.
   A server MUST ensure that every 'href' value within a Multi-Status
   response uses the same format.

   WebDAV only uses one form of relative reference in its extensions,
   the absolute path.

      Simple-ref = absolute-URI | (path-absolute ["?" query])

   The absolute-URI, path-absolute and query productions are defined in
   Sections 4.3, 3.3, and 3.4 of [RFC3986].

   Within Simple-ref productions, senders MUST NOT:

   o use dot-segments ("." or ".."), or

   o have prefixes that do not match the Request-URI (using the
      comparison rules defined in Section 3.2.3 of [RFC2616]).

   Identifiers for collections SHOULD end in a '/' character.

8.3.1. Example - Correct URL Handling

Consider the collection http://example.com/sample/ with the internal
member URL http://example.com/sample/a%20test and the PROPFIND
request below:

>>Request:

```
PROPFIND /sample/ HTTP/1.1
Host: example.com
Depth: 1
```

In this case, the server should return two 'href' elements containing
either

o   'http://example.com/sample/' and
    'http://example.com/sample/a%20test', or

o   '/sample/' and '/sample/a%20test'

Note that even though the server may be storing the member resource
internally as 'a test', it has to be percent-encoded when used inside
a URI reference (see [Section 2.1 of [RFC3986]](#)). Also note that a
legal URI may still contain characters that need to be escaped within
XML character data, such as the ampersand character.

[8.4](#). Required Bodies in Requests

Some of these new methods do not define bodies. Servers MUST examine
all requests for a body, even when a body was not expected. In cases
where a request body is present but would be ignored by a server, the
server MUST reject the request with 415 (Unsupported Media Type).
This informs the client (which may have been attempting to use an
extension) that the body could not be processed as the client
intended.

[8.5](#). HTTP Headers for Use in WebDAV

HTTP defines many headers that can be used in WebDAV requests and
responses. Not all of these are appropriate in all situations and
some interactions may be undefined. Note that HTTP 1.1 requires the

Date header in all responses if possible (see Section 14.18,
[RFC2616]).

The server MUST do authorization checks before checking any HTTP
conditional header.

8.6. ETag

HTTP 1.1 recommends the use of ETags rather than modification dates,
for cache control, and there are even stronger reasons to prefer
ETags for authoring. Correct use of ETags is even more important in
a distributed authoring environment, because ETags are necessary
along with locks to avoid the lost-update problem. A client might
fail to renew a lock, for example, when the lock times out and the
client is accidentally offline or in the middle of a long upload.
When a client fails to renew the lock, it's quite possible the
resource can still be relocked and the user can go on editing, as
long as no changes were made in the meantime. ETags are required for
the client to be able to distinguish this case. Otherwise, the

client is forced to ask the user whether to overwrite the resource on
the server without even being able to tell the user if it has
changed. Timestamps do not solve this problem nearly as well as
ETags.

Strong ETags are much more useful for authoring use cases than weak
ETags (see Section 13.3.3 of [RFC2616]). Semantic equivalence can be
a useful concept but that depends on the document type and the
application type, and interoperability might require some agreement
or standard outside the scope of this specification and HTTP. Note
also that weak ETags have certain restrictions in HTTP, e.g., these
cannot be used in If-Match headers.

Note that the meaning of an ETag in a PUT response is not clearly
defined either in this document or in RFC 2616 (i.e., whether the
ETag means that the resource is octet-for-octet equivalent to the
body of the PUT request, or whether the server could have made minor
changes in the formatting or content of the document upon storage).
This is an HTTP issue, not purely a WebDAV issue.

Because clients may be forced to prompt users or throw away changed
content if the ETag changes, a WebDAV server SHOULD NOT change the

ETag (or the Last-Modified time) for a resource that has an unchanged
body and location. The ETag represents the state of the body or
contents of the resource. There is no similar way to tell if
properties have changed.

8.7. Including Error Response Bodies

   HTTP and WebDAV did not use the bodies of most error responses for
   machine-parsable information until the specification for Versioning
   Extensions to WebDAV introduced a mechanism to include more specific
   information in the body of an error response (Section 1.6 of
   [RFC3253]). The error body mechanism is appropriate to use with any
   error response that may take a body but does not already have a body
   defined. The mechanism is particularly appropriate when a status
   code can mean many things (for example, 400 Bad Request can mean
   required headers are missing, headers are incorrectly formatted, or
   much more). This error body mechanism is covered in Section 16.

8.8. Impact of Namespace Operations on Cache Validators

   Note that the HTTP response headers "Etag" and "Last-Modified" (see
   [RFC2616], Sections 14.19 and 14.29) are defined per URL (not per
   resource), and are used by clients for caching. Therefore servers
   must ensure that executing any operation that affects the URL
   namespace (such as COPY, MOVE, DELETE, PUT, or MKCOL) does preserve
   their semantics, in particular:

Dusseault   Standards Track   [Page 34]

   o For any given URL, the "Last-Modified" value MUST increment every
      time the representation returned upon GET changes (within the
      limits of timestamp resolution).

   o For any given URL, an "ETag" value MUST NOT be reused for
      different representations returned by GET.

   In practice this means that servers

   o might have to increment "Last-Modified" timestamps for every
      resource inside the destination namespace of a namespace operation
      unless it can do so more selectively, and

   o similarly, might have to re-assign "ETag" values for these
      resources (unless the server allocates entity tags in a way so

that they are unique across the whole URL namespace managed by the
server).

Note that these considerations also apply to specific use cases, such
as using PUT to create a new resource at a URL that has been mapped
before, but has been deleted since then.

Finally, WebDAV properties (such as DAV:getetag and DAV:
getlastmodified) that inherit their semantics from HTTP headers must
behave accordingly.

9. HTTP Methods for Distributed Authoring

9.1. PROPFIND Method

The PROPFIND method retrieves properties defined on the resource
identified by the Request-URI, if the resource does not have any
internal members, or on the resource identified by the Request-URI
and potentially its member resources, if the resource is a collection
that has internal member URLs. All DAV-compliant resources MUST
support the PROPFIND method and the propfind XML element
(Section 14.20) along with all XML elements defined for use with that
element.

A client MUST submit a Depth header with a value of "0", "1", or
"infinity" with a PROPFIND request. Servers MUST support "0" and "1"
depth requests on WebDAV-compliant resources and SHOULD support
"infinity" requests. In practice, support for infinite-depth
requests MAY be disabled, due to the performance and security
concerns associated with this behavior. Servers SHOULD treat a
request without a Depth header as if a "Depth: infinity" header was
included.

A client may submit a 'propfind' XML element in the body of the
request method describing what information is being requested. It is
possible to:

o Request particular property values, by naming the properties
   desired within the 'prop' element (the ordering of properties in
   here MAY be ignored by the server),

o Request property values for those properties defined in this
      specification (at a minimum) plus dead properties, by using the
      'allprop' element (the 'include' element can be used with
      'allprop' to instruct the server to also include additional live
      properties that may not have been returned otherwise),

   o Request a list of names of all the properties defined on the
      resource, by using the 'propname' element.

   A client may choose not to submit a request body. An empty PROPFIND
   request body MUST be treated as if it were an 'allprop' request.

   Note that 'allprop' does not return values for all live properties.
   WebDAV servers increasingly have expensively-calculated or lengthy
   properties (see [RFC3253] and [RFC3744]) and do not return all
   properties already. Instead, WebDAV clients can use propname
   requests to discover what live properties exist, and request named
   properties when retrieving values. For a live property defined
   elsewhere, that definition can specify whether or not that live
   property would be returned in 'allprop' requests.

   All servers MUST support returning a response of content type text/
   xml or application/xml that contains a multistatus XML element that
   describes the results of the attempts to retrieve the various
   properties.

   If there is an error retrieving a property, then a proper error
   result MUST be included in the response. A request to retrieve the
   value of a property that does not exist is an error and MUST be noted
   with a 'response' XML element that contains a 404 (Not Found) status
   value.

   Consequently, the 'multistatus' XML element for a collection resource
   MUST include a 'response' XML element for each member URL of the
   collection, to whatever depth was requested. It SHOULD NOT include
   any 'response' elements for resources that are not WebDAV-compliant.
   Each 'response' element MUST contain an 'href' element that contains
   the URL of the resource on which the properties in the prop XML
   element are defined. Results for a PROPFIND on a collection resource
   are returned as a flat list whose order of entries is not

   significant. Note that a resource may have only one value for a

property of a given name, so the property may only show up once in
PROPFIND responses.

Properties may be subject to access control. In the case of
'allprop' and 'propname' requests, if a principal does not have the
right to know whether a particular property exists, then the property
MAY be silently excluded from the response.

Some PROPFIND results MAY be cached, with care, as there is no cache
validation mechanism for most properties. This method is both safe
and idempotent (see Section 9.1 of [RFC2616]).

9.1.1. PROPFIND Status Codes

This section, as with similar sections for other methods, provides
some guidance on error codes and preconditions or postconditions
(defined in Section 16) that might be particularly useful with
PROPFIND.

403 Forbidden - A server MAY reject PROPFIND requests on collections
with depth header of "Infinity", in which case it SHOULD use this
error with the precondition code 'propfind-finite-depth' inside the
error body.

9.1.2. Status Codes for Use in 'propstat' Element

In PROPFIND responses, information about individual properties is
returned inside 'propstat' elements (see Section 14.22), each
containing an individual 'status' element containing information
about the properties appearing in it. The list below summarizes the
most common status codes used inside 'propstat'; however, clients
should be prepared to handle other 2/3/4/5xx series status codes as
well.

200 OK - A property exists and/or its value is successfully returned.

401 Unauthorized - The property cannot be viewed without appropriate
authorization.

403 Forbidden - The property cannot be viewed regardless of
authentication.

404 Not Found - The property does not exist.

9.1.3. Example - Retrieving Named Properties

   >>Request

```
PROPFIND /file HTTP/1.1
Host: www.example.com
Content-type: application/xml; charset="utf-8"
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<D:propfind xmlns:D="DAV:">
  <D:prop xmlns:R="http://ns.example.com/boxschema/">
    <R:bigbox/>
    <R:author/>
    <R:DingALing/>
    <R:Random/>
  </D:prop>
</D:propfind>
```

   >>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: application/xml; charset="utf-8"
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<D:multistatus xmlns:D="DAV:">
  <D:response xmlns:R="http://ns.example.com/boxschema/">
    <D:href>http://www.example.com/file</D:href>
    <D:propstat>
      <D:prop>
        <R:bigbox>
          <R:BoxType>Box type A</R:BoxType>
        </R:bigbox>
        <R:author>
          <R:Name>J.J. Johnson</R:Name>
        </R:author>
      </D:prop>
      <D:status>HTTP/1.1 200 OK</D:status>
    </D:propstat>
    <D:propstat>
      <D:prop><R:DingALing/><R:Random/></D:prop>
      <D:status>HTTP/1.1 403 Forbidden</D:status>
      <D:responsedescription> The user does not have access to the
DingALing property.
```

```
            </D:responsedescription>
          </D:propstat>
```

```
      </D:response>
      <D:responsedescription> There has been an access violation error.
      </D:responsedescription>
    </D:multistatus>
```

   In this example, PROPFIND is executed on a non-collection resource
   http://www.example.com/file. The propfind XML element specifies the
   name of four properties whose values are being requested. In this
   case, only two properties were returned, since the principal issuing
   the request did not have sufficient access rights to see the third
   and fourth properties.

9.1.4. Example - Using 'propname' to Retrieve All Property Names

   >>Request

```
      PROPFIND /container/ HTTP/1.1
      Host: www.example.com
      Content-Type: application/xml; charset="utf-8"
      Content-Length: xxxx

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

   >>Response

```
      HTTP/1.1 207 Multi-Status
      Content-Type: application/xml; charset="utf-8"
      Content-Length: xxxx

      <?xml version="1.0" encoding="utf-8" ?>
      <multistatus xmlns="DAV:">
        <response>
          <href>http://www.example.com/container/</href>
          <propstat>
```

```
                <prop xmlns:R="http://ns.example.com/boxschema/">
                  <R:bigbox/>
                  <R:author/>
                  <creationdate/>
                  <displayname/>
                  <resourcetype/>
                  <supportedlock/>
                </prop>
                <status>HTTP/1.1 200 OK</status>
```

```
            </propstat>
          </response>
          <response>
            <href>http://www.example.com/container/front.html</href>
            <propstat>
              <prop xmlns:R="http://ns.example.com/boxschema/">
                <R:bigbox/>
                <creationdate/>
                <displayname/>
                <getcontentlength/>
                <getcontenttype/>
                <getetag/>
                <getlastmodified/>
                <resourcetype/>
                <supportedlock/>
              </prop>
              <status>HTTP/1.1 200 OK</status>
            </propstat>
          </response>
        </multistatus>
```

   In this example, PROPFIND is invoked on the collection resource
   http://www.example.com/container/, with a propfind XML element
   containing the propname XML element, meaning the name of all
   properties should be returned. Since no Depth header is present, it
   assumes its default value of "infinity", meaning the name of the
   properties on the collection and all its descendants should be
   returned.

   Consistent with the previous example, resource
   http://www.example.com/container/ has six properties defined on it:
   bigbox and author in the "http://ns.example.com/boxschema/"

namespace, and creationdate, displayname, resourcetype, and
supportedlock in the "DAV:" namespace.

The resource http://www.example.com/container/index.html, a member of
the "container" collection, has nine properties defined on it, bigbox
in the "http://ns.example.com/boxschema/" namespace and creationdate,
displayname, getcontentlength, getcontenttype, getetag,
getlastmodified, resourcetype, and supportedlock in the "DAV:"
namespace.

This example also demonstrates the use of XML namespace scoping and
the default namespace. Since the "xmlns" attribute does not contain
a prefix, the namespace applies by default to all enclosed elements.
Hence, all elements that do not explicitly state the namespace to
which they belong are members of the "DAV:" namespace.

[9.1.5](9.1.5).   Example - Using So-called 'allprop'

   Note that 'allprop', despite its name, which remains for backward-
   compatibility, does not return every property, but only dead
   properties and the live properties defined in this specification.

   >>Request

     PROPFIND /container/ HTTP/1.1
     Host: www.example.com
     Depth: 1
     Content-Type: application/xml; charset="utf-8"
     Content-Length: xxxx

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

   >>Response

     HTTP/1.1 207 Multi-Status
     Content-Type: application/xml; charset="utf-8"
     Content-Length: xxxx

```
        <?xml version="1.0" encoding="utf-8" ?>
        <D:multistatus xmlns:D="DAV:">
          <D:response>
            <D:href>/container/</D:href>
            <D:propstat>
              <D:prop xmlns:R="http://ns.example.com/boxschema/">
                <R:bigbox><R:BoxType>Box type A</R:BoxType></R:bigbox>
                <R:author><R:Name>Hadrian</R:Name></R:author>
                <D:creationdate>1997-12-01T17:42:21-08:00</D:creationdate>
                <D:displayname>Example collection</D:displayname>
                <D:resourcetype><D:collection/></D:resourcetype>
                <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:response>
            <D:href>/container/front.html</D:href>
            <D:propstat>
              <D:prop xmlns:R="http://ns.example.com/boxschema/">
                <R:bigbox><R:BoxType>Box type B</R:BoxType>
                </R:bigbox>
                <D:creationdate>1997-12-01T18:27:21-08:00</D:creationdate>
                <D:displayname>Example HTML resource</D:displayname>
                <D:getcontentlength>4525</D:getcontentlength>
                <D:getcontenttype>text/html</D:getcontenttype>
                <D:getetag>"zzyzx"</D:getetag>
                <D:getlastmodified
                  >Mon, 12 Jan 1998 09:25:56 GMT</D:getlastmodified>
                <D:resourcetype/>
                <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>
```

   In this example, PROPFIND was invoked on the resource
   http://www.example.com/container/ with a Depth header of 1, meaning
   the request applies to the resource and its children, and a propfind
   XML element containing the allprop XML element, meaning the request
   should return the name and value of all the dead properties defined
   on the resources, plus the name and value of all the properties
   defined in this specification. This example illustrates the use of
   relative references in the 'href' elements of the response.

   The resource http://www.example.com/container/ has six properties
   defined on it: 'bigbox' and 'author in the
   "http://ns.example.com/boxschema/" namespace, DAV:creationdate, DAV:
   displayname, DAV:resourcetype, and DAV:supportedlock.

   The last four properties are WebDAV-specific, defined in [Section 15](#).
   Since GET is not supported on this resource, the get* properties
   (e.g., DAV:getcontentlength) are not defined on this resource. The
   WebDAV-specific properties assert that "container" was created on
   December 1, 1997, at 5:42:21PM, in a time zone 8 hours west of GMT
   (DAV:creationdate), has a name of "Example collection" (DAV:
   displayname), a collection resource type (DAV:resourcetype), and
   supports exclusive write and shared write locks (DAV:supportedlock).

   The resource http://www.example.com/container/front.html has nine
   properties defined on it:

'bigbox' in the "http://ns.example.com/boxschema/" namespace (another
instance of the "bigbox" property type), DAV:creationdate, DAV:
displayname, DAV:getcontentlength, DAV:getcontenttype, DAV:getetag,
DAV:getlastmodified, DAV:resourcetype, and DAV:supportedlock.

The DAV-specific properties assert that "front.html" was created on
December 1, 1997, at 6:27:21PM, in a time zone 8 hours west of GMT
(DAV:creationdate), has a name of "Example HTML resource" (DAV:
displayname), a content length of 4525 bytes (DAV:getcontentlength),
a MIME type of "text/html" (DAV:getcontenttype), an entity tag of
"zzyzx" (DAV:getetag), was last modified on Monday, January 12, 1998,
at 09:25:56 GMT (DAV:getlastmodified), has an empty resource type,
meaning that it is not a collection (DAV:resourcetype), and supports
both exclusive write and shared write locks (DAV:supportedlock).

9.1.6. Example - Using 'allprop' with 'include'

   >>Request

   PROPFIND /mycol/ HTTP/1.1
   Host: www.example.com
   Depth: 1
   Content-Type: application/xml; charset="utf-8"
   Content-Length: xxxx

   <?xml version="1.0" encoding="utf-8" ?>
   <D:propfind xmlns:D="DAV:">
     <D:allprop/>
     <D:include>
       <D:supported-live-property-set/>
       <D:supported-report-set/>
     </D:include>
   </D:propfind>

   In this example, PROPFIND is executed on the resource
   http://www.example.com/mycol/ and its internal member resources. The
   client requests the values of all live properties defined in this
   specification, plus all dead properties, plus two more live

properties defined in [RFC3253]. The response is not shown.

9.2. PROPPATCH Method

The PROPPATCH method processes instructions specified in the request
body to set and/or remove properties defined on the resource
identified by the Request-URI.

All DAV-compliant resources MUST support the PROPPATCH method and
MUST process instructions that are specified using the
propertyupdate, set, and remove XML elements. Execution of the
directives in this method is, of course, subject to access control
constraints. DAV-compliant resources SHOULD support the setting of
arbitrary dead properties.

The request message body of a PROPPATCH method MUST contain the
propertyupdate XML element.

Servers MUST process PROPPATCH instructions in document order (an
exception to the normal rule that ordering is irrelevant).
Instructions MUST either all be executed or none executed. Thus, if
any error occurs during processing, all executed instructions MUST be
undone and a proper error result returned. Instruction processing
details can be found in the definition of the set and remove
instructions in Sections 14.23 and 14.26.

If a server attempts to make any of the property changes in a
PROPPATCH request (i.e., the request is not rejected for high-level
errors before processing the body), the response MUST be a Multi-
Status response as described in Section 9.2.1.

This method is idempotent, but not safe (see Section 9.1 of
[RFC2616]). Responses to this method MUST NOT be cached.

9.2.1. Status Codes for Use in 'propstat' Element

In PROPPATCH responses, information about individual properties is
returned inside 'propstat' elements (see Section 14.22), each
containing an individual 'status' element containing information
about the properties appearing in it. The list below summarizes the
most common status codes used inside 'propstat'; however, clients
should be prepared to handle other 2/3/4/5xx series status codes as
well.

   200 (OK) - The property set or change succeeded. Note that if this
   appears for one property, it appears for every property in the
   response, due to the atomicity of PROPPATCH.

   403 (Forbidden) - The client, for reasons the server chooses not to
   specify, cannot alter one of the properties.

   403 (Forbidden): The client has attempted to set a protected
   property, such as DAV:getetag. If returning this error, the server
   SHOULD use the precondition code 'cannot-modify-protected-property'
   inside the response body.

   409 (Conflict) - The client has provided a value whose semantics are
   not appropriate for the property.

   424 (Failed Dependency) - The property change could not be made
   because of another property change that failed.

   507 (Insufficient Storage) - The server did not have sufficient space
   to record the property.

[9.2.2](#). Example - PROPPATCH

   >>Request

```
   PROPPATCH /bar.html HTTP/1.1
   Host: www.example.com
   Content-Type: application/xml; charset="utf-8"
   Content-Length: xxxx

   <?xml version="1.0" encoding="utf-8" ?>
   <D:propertyupdate xmlns:D="DAV:"
           xmlns:Z="http://ns.example.com/standards/z39.50/">
     <D:set>
       <D:prop>
         <Z:Authors>
           <Z:Author>Jim Whitehead</Z:Author>
           <Z:Author>Roy Fielding</Z:Author>
         </Z:Authors>
       </D:prop>
     </D:set>
     <D:remove>
       <D:prop><Z:Copyright-Owner/></D:prop>
     </D:remove>
   </D:propertyupdate>
```

```
>>Response

  HTTP/1.1 207 Multi-Status
  Content-Type: application/xml; charset="utf-8"
  Content-Length: xxxx

  <?xml version="1.0" encoding="utf-8" ?>
  <D:multistatus xmlns:D="DAV:"
          xmlns:Z="http://ns.example.com/standards/z39.50/">
    <D:response>
      <D:href>http://www.example.com/bar.html</D:href>
      <D:propstat>
        <D:prop><Z:Authors/></D:prop>
        <D:status>HTTP/1.1 424 Failed Dependency</D:status>
      </D:propstat>
      <D:propstat>
        <D:prop><Z:Copyright-Owner/></D:prop>
        <D:status>HTTP/1.1 409 Conflict</D:status>
      </D:propstat>
      <D:responsedescription> Copyright Owner cannot be deleted or
        altered.</D:responsedescription>
    </D:response>
  </D:multistatus>
```

   In this example, the client requests the server to set the value of
   the "Authors" property in the
   "http://ns.example.com/standards/z39.50/" namespace, and to remove
   the property "Copyright-Owner" in the same namespace. Since the
   Copyright-Owner property could not be removed, no property
   modifications occur. The 424 (Failed Dependency) status code for the
   Authors property indicates this action would have succeeded if it
   were not for the conflict with removing the Copyright-Owner property.

9.3. MKCOL Method

   MKCOL creates a new collection resource at the location specified by
   the Request-URI. If the Request-URI is already mapped to a resource,
   then the MKCOL MUST fail. During MKCOL processing, a server MUST
   make the Request-URI an internal member of its parent collection,
   unless the Request-URI is "/". If no such ancestor exists, the

method MUST fail. When the MKCOL operation creates a new collection
resource, all ancestors MUST already exist, or the method MUST fail
with a 409 (Conflict) status code. For example, if a request to
create collection /a/b/c/d/ is made, and /a/b/c/ does not exist, the
request must fail.

When MKCOL is invoked without a request body, the newly created
collection SHOULD have no members.

A MKCOL request message may contain a message body. The precise
behavior of a MKCOL request when the body is present is undefined,
but limited to creating collections, members of a collection, bodies
of members, and properties on the collections or members. If the
server receives a MKCOL request entity type it does not support or
understand, it MUST respond with a 415 (Unsupported Media Type)
status code. If the server decides to reject the request based on
the presence of an entity or the type of an entity, it should use the
415 (Unsupported Media Type) status code.

This method is idempotent, but not safe (see [Section 9.1 of
   [RFC2616]](#)). Responses to this method MUST NOT be cached.

9.3.1. MKCOL Status Codes

In addition to the general status codes possible, the following
status codes have specific applicability to MKCOL:

201 (Created) - The collection was created.

403 (Forbidden) - This indicates at least one of two conditions: 1)
the server does not allow the creation of collections at the given
location in its URL namespace, or 2) the parent collection of the
Request-URI exists but cannot accept members.

405 (Method Not Allowed) - MKCOL can only be executed on an unmapped
URL.

409 (Conflict) - A collection cannot be made at the Request-URI until
one or more intermediate collections have been created. The server
MUST NOT create those intermediate collections automatically.

415 (Unsupported Media Type) - The server does not support the

request body type (although bodies are legal on MKCOL requests, since
this specification doesn't define any, the server is likely not to
support any given body type).

507 (Insufficient Storage) - The resource does not have sufficient
space to record the state of the resource after the execution of this
method.

9.3.2. Example - MKCOL

This example creates a collection called /webdisc/xfiles/ on the
server www.example.com.

>>Request

    MKCOL /webdisc/xfiles/ HTTP/1.1
    Host: www.example.com

>>Response

    HTTP/1.1 201 Created

9.4. GET, HEAD for Collections

The semantics of GET are unchanged when applied to a collection,
since GET is defined as, "retrieve whatever information (in the form
of an entity) is identified by the Request-URI" [RFC2616]. GET, when
applied to a collection, may return the contents of an "index.html"
resource, a human-readable view of the contents of the collection, or
something else altogether. Hence, it is possible that the result of
a GET on a collection will bear no correlation to the membership of
the collection.

Similarly, since the definition of HEAD is a GET without a response
message body, the semantics of HEAD are unmodified when applied to
collection resources.

9.5. POST for Collections

Since by definition the actual function performed by POST is
determined by the server and often depends on the particular
resource, the behavior of POST when applied to collections cannot be
meaningfully modified because it is largely undefined. Thus, the
semantics of POST are unmodified when applied to a collection.

9.6. DELETE Requirements

DELETE is defined in [RFC2616], Section 9.7, to "delete the resource
identified by the Request-URI". However, WebDAV changes some DELETE
handling requirements.

A server processing a successful DELETE request:

   MUST destroy locks rooted on the deleted resource

   MUST remove the mapping from the Request-URI to any resource.

Thus, after a successful DELETE operation (and in the absence of
other actions), a subsequent GET/HEAD/PROPFIND request to the target
Request-URI MUST return 404 (Not Found).

9.6.1. DELETE for Collections

The DELETE method on a collection MUST act as if a "Depth: infinity"
header was used on it. A client MUST NOT submit a Depth header with
a DELETE on a collection with any value but infinity.

DELETE instructs that the collection specified in the Request-URI and
all resources identified by its internal member URLs are to be
deleted.

If any resource identified by a member URL cannot be deleted, then
all of the member's ancestors MUST NOT be deleted, so as to maintain
URL namespace consistency.

Any headers included with DELETE MUST be applied in processing every
resource to be deleted.

When the DELETE method has completed processing, it MUST result in a
consistent URL namespace.

If an error occurs deleting a member resource (a resource other than
the resource identified in the Request-URI), then the response can be
a 207 (Multi-Status). Multi-Status is used here to indicate which
internal resources could NOT be deleted, including an error code,
which should help the client understand which resources caused the
failure. For example, the Multi-Status body could include a response
with status 423 (Locked) if an internal resource was locked.

The server MAY return a 4xx status response, rather than a 207, if
the request failed completely.

424 (Failed Dependency) status codes SHOULD NOT be in the 207 (Multi-
Status) response for DELETE. They can be safely left out because the
client will know that the ancestors of a resource could not be
deleted when the client receives an error for the ancestor's progeny.
Additionally, 204 (No Content) errors SHOULD NOT be returned in the
207 (Multi-Status). The reason for this prohibition is that 204 (No
Content) is the default success code.

9.6.2. Example - DELETE

   >>Request

     DELETE /container/ HTTP/1.1
     Host: www.example.com

   >>Response

     HTTP/1.1 207 Multi-Status
     Content-Type: application/xml; charset="utf-8"
     Content-Length: xxxx

     <?xml version="1.0" encoding="utf-8" ?>
     <d:multistatus xmlns:d="DAV:">
       <d:response>
         <d:href>http://www.example.com/container/resource3</d:href>
         <d:status>HTTP/1.1 423 Locked</d:status>
         <d:error><d:lock-token-submitted/></d:error>

```
        </d:response>
      </d:multistatus>
```

   In this example, the attempt to delete
   http://www.example.com/container/resource3 failed because it is
   locked, and no lock token was submitted with the request.
   Consequently, the attempt to delete http://www.example.com/container/
   also failed. Thus, the client knows that the attempt to delete
   http://www.example.com/container/ must have also failed since the
   parent cannot be deleted unless its child has also been deleted.
   Even though a Depth header has not been included, a depth of infinity
   is assumed because the method is on a collection.

9.7. PUT Requirements

9.7.1. PUT for Non-Collection Resources

   A PUT performed on an existing resource replaces the GET response
   entity of the resource. Properties defined on the resource may be
   recomputed during PUT processing but are not otherwise affected. For
   example, if a server recognizes the content type of the request body,
   it may be able to automatically extract information that could be
   profitably exposed as properties.

   A PUT that would result in the creation of a resource without an
   appropriately scoped parent collection MUST fail with a 409
   (Conflict).

   A PUT request allows a client to indicate what media type an entity
   body has, and whether it should change if overwritten. Thus, a
   client SHOULD provide a Content-Type for a new resource if any is
   known. If the client does not provide a Content-Type for a new
   resource, the server MAY create a resource with no Content-Type
   assigned, or it MAY attempt to assign a Content-Type.

   Note that although a recipient ought generally to treat metadata
   supplied with an HTTP request as authoritative, in practice there's
   no guarantee that a server will accept client-supplied metadata
   (e.g., any request header beginning with "Content-"). Many servers
   do not allow configuring the Content-Type on a per-resource basis in

the first place. Thus, clients can't always rely on the ability to
directly influence the content type by including a Content-Type
request header.

9.7.2. PUT for Collections

This specification does not define the behavior of the PUT method for
existing collections. A PUT request to an existing collection MAY be
treated as an error (405 Method Not Allowed).

The MKCOL method is defined to create collections.

9.8. COPY Method

The COPY method creates a duplicate of the source resource identified
by the Request-URI, in the destination resource identified by the URI
in the Destination header. The Destination header MUST be present.
The exact behavior of the COPY method depends on the type of the
source resource.

All WebDAV-compliant resources MUST support the COPY method.
However, support for the COPY method does not guarantee the ability
to copy a resource. For example, separate programs may control
resources on the same server. As a result, it may not be possible to
copy a resource to a location that appears to be on the same server.

This method is idempotent, but not safe (see Section 9.1 of
[RFC2616]). Responses to this method MUST NOT be cached.

9.8.1. COPY for Non-collection Resources

When the source resource is not a collection, the result of the COPY
method is the creation of a new resource at the destination whose
state and behavior match that of the source resource as closely as
possible. Since the environment at the destination may be different
than at the source due to factors outside the scope of control of the
server, such as the absence of resources required for correct
operation, it may not be possible to completely duplicate the
behavior of the resource at the destination. Subsequent alterations
to the destination resource will not modify the source resource.
Subsequent alterations to the source resource will not modify the
destination resource.

. COPY for Properties

   After a successful COPY invocation, all dead properties on the source
   resource SHOULD be duplicated on the destination resource. Live
   properties described in this document SHOULD be duplicated as
   identically behaving live properties at the destination resource, but
   not necessarily with the same values. Servers SHOULD NOT convert
   live properties into dead properties on the destination resource,
   because clients may then draw incorrect conclusions about the state
   or functionality of a resource. Note that some live properties are
   defined such that the absence of the property has a specific meaning
   (e.g., a flag with one meaning if present, and the opposite if
   absent), and in these cases, a successful COPY might result in the
   property being reported as "Not Found" in subsequent requests.

   When the destination is an unmapped URL, a COPY operation creates a
   new resource much like a PUT operation does. Live properties that
   are related to resource creation (such as DAV:creationdate) should
   have their values set accordingly.

. COPY for Collections

   The COPY method on a collection without a Depth header MUST act as if
   a Depth header with value "infinity" was included. A client may
   submit a Depth header on a COPY on a collection with a value of "0"
   or "infinity". Servers MUST support the "0" and "infinity" Depth
   header behaviors on WebDAV-compliant resources.

   An infinite-depth COPY instructs that the collection resource
   identified by the Request-URI is to be copied to the location
   identified by the URI in the Destination header, and all its internal
   member resources are to be copied to a location relative to it,
   recursively through all levels of the collection hierarchy. Note
   that an infinite-depth COPY of /A/ into /A/B/ could lead to infinite
   recursion if not handled correctly.

   A COPY of "Depth: 0" only instructs that the collection and its
   properties, but not resources identified by its internal member URLs,
   are to be copied.

   Any headers included with a COPY MUST be applied in processing every
   resource to be copied with the exception of the Destination header.

   The Destination header only specifies the destination URI for the
   Request-URI. When applied to members of the collection identified by
   the Request-URI, the value of Destination is to be modified to
   reflect the current location in the hierarchy. So, if the Request-
   URI is /a/ with Host header value http://example.com/ and the

   Destination is http://example.com/b/, then when
   http://example.com/a/c/d is processed, it must use a Destination of
   http://example.com/b/c/d.

   When the COPY method has completed processing, it MUST have created a
   consistent URL namespace at the destination (see [Section 5.1](#) for the
   definition of namespace consistency). However, if an error occurs
   while copying an internal collection, the server MUST NOT copy any
   resources identified by members of this collection (i.e., the server
   must skip this subtree), as this would create an inconsistent
   namespace. After detecting an error, the COPY operation SHOULD try
   to finish as much of the original copy operation as possible (i.e.,
   the server should still attempt to copy other subtrees and their
   members that are not descendants of an error-causing collection).

   So, for example, if an infinite-depth copy operation is performed on
   collection /a/, which contains collections /a/b/ and /a/c/, and an
   error occurs copying /a/b/, an attempt should still be made to copy
   /a/c/. Similarly, after encountering an error copying a non-
   collection resource as part of an infinite-depth copy, the server
   SHOULD try to finish as much of the original copy operation as
   possible.

   If an error in executing the COPY method occurs with a resource other
   than the resource identified in the Request-URI, then the response
   MUST be a 207 (Multi-Status), and the URL of the resource causing the
   failure MUST appear with the specific error.

   The 424 (Failed Dependency) status code SHOULD NOT be returned in the
   207 (Multi-Status) response from a COPY method. These responses can
   be safely omitted because the client will know that the progeny of a
   resource could not be copied when the client receives an error for
   the parent. Additionally, 201 (Created)/204 (No Content) status
   codes SHOULD NOT be returned as values in 207 (Multi-Status)
   responses from COPY methods. They, too, can be safely omitted
   because they are the default success codes.

[9.8.4](#). COPY and Overwriting Destination Resources

   If a COPY request has an Overwrite header with a value of "F", and a
   resource exists at the Destination URL, the server MUST fail the
   request.

When a server executes a COPY request and overwrites a destination
resource, the exact behavior MAY depend on many factors, including
WebDAV extension capabilities (see particularly [RFC3253]). For

example, when an ordinary resource is overwritten, the server could
delete the target resource before doing the copy, or could do an in-
place overwrite to preserve live properties.

When a collection is overwritten, the membership of the destination
collection after the successful COPY request MUST be the same
membership as the source collection immediately before the COPY.
Thus, merging the membership of the source and destination
collections together in the destination is not a compliant behavior.

In general, if clients require the state of the destination URL to be
wiped out prior to a COPY (e.g., to force live properties to be
reset), then the client could send a DELETE to the destination before
the COPY request to ensure this reset.

9.8.5. Status Codes

In addition to the general status codes possible, the following
status codes have specific applicability to COPY:

201 (Created) - The source resource was successfully copied. The
COPY operation resulted in the creation of a new resource.

204 (No Content) - The source resource was successfully copied to a
preexisting destination resource.

207 (Multi-Status) - Multiple resources were to be affected by the
COPY, but errors on some of them prevented the operation from taking
place. Specific error messages, together with the most appropriate
of the source and destination URLs, appear in the body of the multi-
status response. For example, if a destination resource was locked
and could not be overwritten, then the destination resource URL
appears with the 423 (Locked) status.

403 (Forbidden) - The operation is forbidden. A special case for

COPY could be that the source and destination resources are the same
resource.

409 (Conflict) - A resource cannot be created at the destination
until one or more intermediate collections have been created. The
server MUST NOT create those intermediate collections automatically.

412 (Precondition Failed) - A precondition header check failed, e.g.,
the Overwrite header is "F" and the destination URL is already mapped
to a resource.

423 (Locked) - The destination resource, or resource within the
destination collection, was locked. This response SHOULD contain the
'lock-token-submitted' precondition element.

502 (Bad Gateway) - This may occur when the destination is on another
server, repository, or URL namespace. Either the source namespace
does not support copying to the destination namespace, or the
destination namespace refuses to accept the resource. The client may
wish to try GET/PUT and PROPFIND/PROPPATCH instead.

507 (Insufficient Storage) - The destination resource does not have
sufficient space to record the state of the resource after the
execution of this method.

9.8.6. Example - COPY with Overwrite

This example shows resource
http://www.example.com/~fielding/index.html being copied to the
location http://www.example.com/users/f/fielding/index.html. The 204
(No Content) status code indicates that the existing resource at the
destination was overwritten.

>>Request

   COPY /~fielding/index.html HTTP/1.1
   Host: www.example.com
   Destination: http://www.example.com/users/f/fielding/index.html

>>Response

   HTTP/1.1 204 No Content

9.8.7. Example - COPY with No Overwrite

The following example shows the same copy operation being performed,
but with the Overwrite header set to "F." A response of 412
(Precondition Failed) is returned because the destination URL is
already mapped to a resource.

>>Request

   COPY /~fielding/index.html HTTP/1.1
   Host: www.example.com
   Destination: http://www.example.com/users/f/fielding/index.html
   Overwrite: F

>>Response

   HTTP/1.1 412 Precondition Failed

9.8.8. Example - COPY of a Collection

>>Request

   COPY /container/ HTTP/1.1
   Host: www.example.com
   Destination: http://www.example.com/othercontainer/
   Depth: infinity

>>Response

   HTTP/1.1 207 Multi-Status
   Content-Type: application/xml; charset="utf-8"
   Content-Length: xxxx

   <?xml version="1.0" encoding="utf-8" ?>

```
          <d:multistatus xmlns:d="DAV:">
            <d:response>
              <d:href>http://www.example.com/othercontainer/R2/</d:href>
              <d:status>HTTP/1.1 423 Locked</d:status>
              <d:error><d:lock-token-submitted/></d:error>
            </d:response>
          </d:multistatus>
```

   The Depth header is unnecessary as the default behavior of COPY on a
   collection is to act as if a "Depth: infinity" header had been
   submitted. In this example, most of the resources, along with the
   collection, were copied successfully. However, the collection R2
   failed because the destination R2 is locked. Because there was an
   error copying R2, none of R2's members were copied. However, no
   errors were listed for those members due to the error minimization
   rules.

[9.9](#). MOVE Method

   The MOVE operation on a non-collection resource is the logical
   equivalent of a copy (COPY), followed by consistency maintenance
   processing, followed by a delete of the source, where all three
   actions are performed in a single operation. The consistency
   maintenance step allows the server to perform updates caused by the
   move, such as updating all URLs, other than the Request-URI that
   identifies the source resource, to point to the new destination
   resource.

   The Destination header MUST be present on all MOVE methods and MUST
   follow all COPY requirements for the COPY part of the MOVE method.
   All WebDAV-compliant resources MUST support the MOVE method.

   Support for the MOVE method does not guarantee the ability to move a
   resource to a particular destination. For example, separate programs
   may actually control different sets of resources on the same server.
   Therefore, it may not be possible to move a resource within a
   namespace that appears to belong to the same server.

   If a resource exists at the destination, the destination resource
   will be deleted as a side-effect of the MOVE operation, subject to
   the restrictions of the Overwrite header.

This method is idempotent, but not safe (see [Section 9.1 of   [RFC2616]](#)). Responses to this method MUST NOT be cached.

9.9.1. MOVE for Properties

Live properties described in this document SHOULD be moved along with the resource, such that the resource has identically behaving live properties at the destination resource, but not necessarily with the same values. Note that some live properties are defined such that the absence of the property has a specific meaning (e.g., a flag with one meaning if present, and the opposite if absent), and in these cases, a successful MOVE might result in the property being reported as "Not Found" in subsequent requests. If the live properties will not work the same way at the destination, the server MAY fail the request.

MOVE is frequently used by clients to rename a file without changing its parent collection, so it's not appropriate to reset all live properties that are set at resource creation. For example, the DAV: creationdate property value SHOULD remain the same after a MOVE.

Dead properties MUST be moved along with the resource.

9.9.2. MOVE for Collections

A MOVE with "Depth: infinity" instructs that the collection identified by the Request-URI be moved to the address specified in the Destination header, and all resources identified by its internal member URLs are to be moved to locations relative to it, recursively through all levels of the collection hierarchy.

The MOVE method on a collection MUST act as if a "Depth: infinity" header was used on it. A client MUST NOT submit a Depth header on a MOVE on a collection with any value but "infinity".

Any headers included with MOVE MUST be applied in processing every resource to be moved with the exception of the Destination header. The behavior of the Destination header is the same as given for COPY on collections.

When the MOVE method has completed processing, it MUST have created a consistent URL namespace at both the source and destination (see

for the definition of namespace consistency). However,
   if an error occurs while moving an internal collection, the server
   MUST NOT move any resources identified by members of the failed
   collection (i.e., the server must skip the error-causing subtree), as
   this would create an inconsistent namespace. In this case, after
   detecting the error, the move operation SHOULD try to finish as much
   of the original move as possible (i.e., the server should still
   attempt to move other subtrees and the resources identified by their
   members that are not descendants of an error-causing collection).
   So, for example, if an infinite-depth move is performed on collection
   /a/, which contains collections /a/b/ and /a/c/, and an error occurs
   moving /a/b/, an attempt should still be made to try moving /a/c/.
   Similarly, after encountering an error moving a non-collection
   resource as part of an infinite-depth move, the server SHOULD try to
   finish as much of the original move operation as possible.

   If an error occurs with a resource other than the resource identified
   in the Request-URI, then the response MUST be a 207 (Multi-Status),
   and the errored resource's URL MUST appear with the specific error.

   The 424 (Failed Dependency) status code SHOULD NOT be returned in the
   207 (Multi-Status) response from a MOVE method. These errors can be
   safely omitted because the client will know that the progeny of a
   resource could not be moved when the client receives an error for the
   parent. Additionally, 201 (Created)/204 (No Content) responses
   SHOULD NOT be returned as values in 207 (Multi-Status) responses from
   a MOVE. These responses can be safely omitted because they are the
   default success codes.

9.9.3. MOVE and the Overwrite Header

   If a resource exists at the destination and the Overwrite header is
   "T", then prior to performing the move, the server MUST perform a
   DELETE with "Depth: infinity" on the destination resource. If the
   Overwrite header is set to "F", then the operation will fail.

9.9.4. Status Codes

In addition to the general status codes possible, the following
status codes have specific applicability to MOVE:

201 (Created) - The source resource was successfully moved, and a new
URL mapping was created at the destination.

204 (No Content) - The source resource was successfully moved to a
URL that was already mapped.

207 (Multi-Status) - Multiple resources were to be affected by the
MOVE, but errors on some of them prevented the operation from taking
place. Specific error messages, together with the most appropriate
of the source and destination URLs, appear in the body of the multi-
status response. For example, if a source resource was locked and
could not be moved, then the source resource URL appears with the 423
(Locked) status.

403 (Forbidden) - Among many possible reasons for forbidding a MOVE
operation, this status code is recommended for use when the source
and destination resources are the same.

409 (Conflict) - A resource cannot be created at the destination
until one or more intermediate collections have been created. The
server MUST NOT create those intermediate collections automatically.
Or, the server was unable to preserve the behavior of the live
properties and still move the resource to the destination (see
'preserved-live-properties' postcondition).

412 (Precondition Failed) - A condition header failed. Specific to
MOVE, this could mean that the Overwrite header is "F" and the
destination URL is already mapped to a resource.

423 (Locked) - The source or the destination resource, the source or
destination resource parent, or some resource within the source or
destination collection, was locked. This response SHOULD contain the
'lock-token-submitted' precondition element.

502 (Bad Gateway) - This may occur when the destination is on another
server and the destination server refuses to accept the resource.
This could also occur when the destination is on another sub-section
of the same server namespace.

9.9.5. Example - MOVE of a Non-Collection

   This example shows resource
   http://www.example.com/~fielding/index.html being moved to the
   location http://www.example.com/users/f/fielding/index.html. The
   contents of the destination resource would have been overwritten if
   the destination URL was already mapped to a resource. In this case,
   since there was nothing at the destination resource, the response
   code is 201 (Created).

   >>Request

     MOVE /~fielding/index.html HTTP/1.1
     Host: www.example.com
     Destination: http://www.example/users/f/fielding/index.html

   >>Response

     HTTP/1.1 201 Created
     Location: http://www.example.com/users/f/fielding/index.html

9.9.6. Example - MOVE of a Collection

   >>Request

     MOVE /container/ HTTP/1.1
     Host: www.example.com
     Destination: http://www.example.com/othercontainer/
     Overwrite: F
     If: (<urn:uuid:fe184f2e-6eec-41d0-c765-01adc56e6bb4>)
        (<urn:uuid:e454f3f3-acdc-452a-56c7-00a5c91e4b77>)

   >>Response

     HTTP/1.1 207 Multi-Status
     Content-Type: application/xml; charset="utf-8"
     Content-Length: xxxx

     <?xml version="1.0" encoding="utf-8" ?>
     <d:multistatus xmlns:d='DAV:'>
       <d:response>
         <d:href>http://www.example.com/othercontainer/C2/</d:href>
         <d:status>HTTP/1.1 423 Locked</d:status>

```
      <d:error><d:lock-token-submitted/></d:error>
    </d:response>
  </d:multistatus>
```

In this example, the client has submitted a number of lock tokens
with the request. A lock token will need to be submitted for every
resource, both source and destination, anywhere in the scope of the
method, that is locked. In this case, the proper lock token was not
submitted for the destination
http://www.example.com/othercontainer/C2/. This means that the
resource /container/C2/ could not be moved. Because there was an
error moving /container/C2/, none of /container/C2's members were
moved. However, no errors were listed for those members due to the
error minimization rules. User agent authentication has previously
occurred via a mechanism outside the scope of the HTTP protocol, in
an underlying transport layer.

9.10. LOCK Method

The following sections describe the LOCK method, which is used to
take out a lock of any access type and to refresh an existing lock.
These sections on the LOCK method describe only those semantics that
are specific to the LOCK method and are independent of the access
type of the lock being requested.

Any resource that supports the LOCK method MUST, at minimum, support
the XML request and response formats defined herein.

This method is neither idempotent nor safe (see [Section 9.1 of
   [RFC2616]](#)). Responses to this method MUST NOT be cached.

9.10.1. Creating a Lock on an Existing Resource

A LOCK request to an existing resource will create a lock on the
resource identified by the Request-URI, provided the resource is not
already locked with a conflicting lock. The resource identified in
the Request-URI becomes the root of the lock. LOCK method requests
to create a new lock MUST have an XML request body. The server MUST
preserve the information provided by the client in the 'owner'
element in the LOCK request. The LOCK request MAY have a Timeout

header.

When a new lock is created, the LOCK response:

o MUST contain a body with the value of the DAV:lockdiscovery
   property in a prop XML element. This MUST contain the full
   information about the lock just granted, while information about
   other (shared) locks is OPTIONAL.

o MUST include the Lock-Token response header with the token
   associated with the new lock.

9.10.2. Refreshing Locks

A lock is refreshed by sending a LOCK request to the URL of a
resource within the scope of the lock. This request MUST NOT have a
body and it MUST specify which lock to refresh by using the 'If'
header with a single lock token (only one lock may be refreshed at a
time). The request MAY contain a Timeout header, which a server MAY
accept to change the duration remaining on the lock to the new value.
A server MUST ignore the Depth header on a LOCK refresh.

If the resource has other (shared) locks, those locks are unaffected
by a lock refresh. Additionally, those locks do not prevent the
named lock from being refreshed.

The Lock-Token header is not returned in the response for a
successful refresh LOCK request, but the LOCK response body MUST
contain the new value for the DAV:lockdiscovery property.

9.10.3. Depth and Locking

The Depth header may be used with the LOCK method. Values other than
0 or infinity MUST NOT be used with the Depth header on a LOCK
method. All resources that support the LOCK method MUST support the
Depth header.

A Depth header of value 0 means to just lock the resource specified
by the Request-URI.

If the Depth header is set to infinity, then the resource specified

in the Request-URI along with all its members, all the way down the
hierarchy, are to be locked. A successful result MUST return a
single lock token. Similarly, if an UNLOCK is successfully executed
on this token, all associated resources are unlocked. Hence, partial
success is not an option for LOCK or UNLOCK. Either the entire
hierarchy is locked or no resources are locked.

If the lock cannot be granted to all resources, the server MUST
return a Multi-Status response with a 'response' element for at least
one resource that prevented the lock from being granted, along with a
suitable status code for that failure (e.g., 403 (Forbidden) or 423
(Locked)). Additionally, if the resource causing the failure was not
the resource requested, then the server SHOULD include a 'response'
element for the Request-URI as well, with a 'status' element
containing 424 Failed Dependency.

If no Depth header is submitted on a LOCK request, then the request
MUST act as if a "Depth:infinity" had been submitted.

9.10.4. Locking Unmapped URLs

A successful LOCK method MUST result in the creation of an empty
resource that is locked (and that is not a collection) when a
resource did not previously exist at that URL. Later on, the lock
may go away but the empty resource remains. Empty resources MUST
then appear in PROPFIND responses including that URL in the response
scope. A server MUST respond successfully to a GET request to an
empty resource, either by using a 204 No Content response, or by
using 200 OK with a Content-Length header indicating zero length

9.10.5. Lock Compatibility Table

The table below describes the behavior that occurs when a lock
request is made on a resource.

Current State	Shared Lock OK	Exclusive Lock OK
None	True	True
Shared Lock	True	False
Exclusive Lock	False	False*

```
              +------------------------+---------------+-------------------+
```

Legend: True = lock may be granted. False = lock MUST NOT be
granted. *=It is illegal for a principal to request the same lock
twice.

The current lock state of a resource is given in the leftmost column,
and lock requests are listed in the first row. The intersection of a
row and column gives the result of a lock request. For example, if a
shared lock is held on a resource, and an exclusive lock is
requested, the table entry is "false", indicating that the lock must
not be granted.

[9.10.6](). LOCK Responses

In addition to the general status codes possible, the following
status codes have specific applicability to LOCK:

200 (OK) - The LOCK request succeeded and the value of the DAV:
lockdiscovery property is included in the response body.

201 (Created) - The LOCK request was to an unmapped URL, the request
succeeded and resulted in the creation of a new resource, and the
value of the DAV:lockdiscovery property is included in the response
body.

409 (Conflict) - A resource cannot be created at the destination
until one or more intermediate collections have been created. The
server MUST NOT create those intermediate collections automatically.

423 (Locked), potentially with 'no-conflicting-lock' precondition
code - There is already a lock on the resource that is not compatible
with the requested lock (see lock compatibility table above).

412 (Precondition Failed), with 'lock-token-matches-request-uri'
precondition code - The LOCK request was made with an If header,
indicating that the client wishes to refresh the given lock.
However, the Request-URI did not fall within the scope of the lock
identified by the token. The lock may have a scope that does not
include the Request-URI, or the lock could have disappeared, or the
token may be invalid.

. Example - Simple Lock Request

   >>Request

```
LOCK /workspace/webdav/proposal.doc HTTP/1.1
Host: example.com
Timeout: Infinite, Second-4100000000
Content-Type: application/xml; charset="utf-8"
Content-Length: xxxx
Authorization: Digest username="ejw",
  realm="ejw@example.com", nonce="...",
  uri="/workspace/webdav/proposal.doc",
  response="...", opaque="..."

<?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>http://example.org/~ejw/contact.html</D:href>
  </D:owner>
</D:lockinfo>
```

   >>Response

```
HTTP/1.1 200 OK
Lock-Token: <urn:uuid:e71d4fae-5dec-22d6-fea5-00a0c91e6be4>
Content-Type: application/xml; charset="utf-8"
Content-Length: xxxx

<?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>
        <D:owner>
          <D:href>http://example.org/~ejw/contact.html</D:href>
        </D:owner>
```

```
            <D:timeout>Second-604800</D:timeout>
            <D:locktoken>
              <D:href
              >urn:uuid:e71d4fae-5dec-22d6-fea5-00a0c91e6be4</D:href>
            </D:locktoken>
            <D:lockroot>
              <D:href
              >http://example.com/workspace/webdav/proposal.doc</D:href>
            </D:lockroot>
          </D:activelock>
        </D:lockdiscovery>
      </D:prop>
```

   This example shows the successful creation of an exclusive write lock
   on resource http://example.com/workspace/webdav/proposal.doc. The
   resource http://example.org/~ejw/contact.html contains contact
   information for the creator of the lock. The server has an activity-
   based timeout policy in place on this resource, which causes the lock
   to automatically be removed after 1 week (604800 seconds). Note that
   the nonce, response, and opaque fields have not been calculated in
   the Authorization request header.

9.10.8. Example - Refreshing a Write Lock

   >>Request

```
     LOCK /workspace/webdav/proposal.doc HTTP/1.1
     Host: example.com
     Timeout: Infinite, Second-4100000000
     If: (<urn:uuid:e71d4fae-5dec-22d6-fea5-00a0c91e6be4>)
     Authorization: Digest username="ejw",
       realm="ejw@example.com", nonce="...",
       uri="/workspace/webdav/proposal.doc",
       response="...", opaque="..."
```

   >>Response

```
HTTP/1.1 200 OK
Content-Type: application/xml; charset="utf-8"
Content-Length: xxxx

<?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>
      <D:owner>
        <D:href>http://example.org/~ejw/contact.html</D:href>
      </D:owner>
      <D:timeout>Second-604800</D:timeout>
      <D:locktoken>
        <D:href
        >urn:uuid:e71d4fae-5dec-22d6-fea5-00a0c91e6be4</D:href>
      </D:locktoken>
      <D:lockroot>
        <D:href
        >http://example.com/workspace/webdav/proposal.doc</D:href>
      </D:lockroot>
    </D:activelock>
  </D:lockdiscovery>
</D:prop>
```

This request would refresh the lock, attempting to reset the timeout
to the new value specified in the timeout header. Notice that the
client asked for an infinite time out but the server choose to ignore
the request. In this example, the nonce, response, and opaque fields
have not been calculated in the Authorization request header.

9.10.9. Example - Multi-Resource Lock Request

   >>Request

```
LOCK /webdav/ HTTP/1.1
Host: example.com
Timeout: Infinite, Second-4100000000
Depth: infinity
Content-Type: application/xml; charset="utf-8"
Content-Length: xxxx
Authorization: Digest username="ejw",
  realm="ejw@example.com", nonce="...",
```

```
      uri="/workspace/webdav/proposal.doc",
      response="...", opaque="..."

   <?xml version="1.0" encoding="utf-8" ?>
   <D:lockinfo xmlns:D="DAV:">
     <D:locktype><D:write/></D:locktype>
     <D:lockscope><D:exclusive/></D:lockscope>
     <D:owner>
       <D:href>http://example.org/~ejw/contact.html</D:href>
     </D:owner>
   </D:lockinfo>

>>Response

   HTTP/1.1 207 Multi-Status
   Content-Type: application/xml; charset="utf-8"
   Content-Length: xxxx

   <?xml version="1.0" encoding="utf-8" ?>
   <D:multistatus xmlns:D="DAV:">
     <D:response>
       <D:href>http://example.com/webdav/secret</D:href>
       <D:status>HTTP/1.1 403 Forbidden</D:status>
     </D:response>
     <D:response>
       <D:href>http://example.com/webdav/</D:href>
       <D:status>HTTP/1.1 424 Failed Dependency</D:status>
     </D:response>
   </D:multistatus>
```

This example shows a request for an exclusive write lock on a
collection and all its children. In this request, the client has
specified that it desires an infinite-length lock, if available,
otherwise a timeout of 4.1 billion seconds, if available. The
request entity body contains the contact information for the
principal taking out the lock -- in this case, a Web page URL.

The error is a 403 (Forbidden) response on the resource
http://example.com/webdav/secret. Because this resource could not be
locked, none of the resources were locked. Note also that the a
'response' element for the Request-URI itself has been included as
required.

In this example, the nonce, response, and opaque fields have not been
calculated in the Authorization request header.

9.11. UNLOCK Method

The UNLOCK method removes the lock identified by the lock token in
the Lock-Token request header. The Request-URI MUST identify a
resource within the scope of the lock.

Note that use of the Lock-Token header to provide the lock token is
not consistent with other state-changing methods, which all require
an If header with the lock token. Thus, the If header is not needed
to provide the lock token. Naturally, when the If header is present,
it has its normal meaning as a conditional header.

For a successful response to this method, the server MUST delete the
lock entirely.

If all resources that have been locked under the submitted lock token
cannot be unlocked, then the UNLOCK request MUST fail.

A successful response to an UNLOCK method does not mean that the
resource is necessarily unlocked. It means that the specific lock
corresponding to the specified token no longer exists.

Any DAV-compliant resource that supports the LOCK method MUST support
the UNLOCK method.

This method is idempotent, but not safe (see Section 9.1 of
   [RFC2616]). Responses to this method MUST NOT be cached.

9.11.1. Status Codes

In addition to the general status codes possible, the following
status codes have specific applicability to UNLOCK:

204 (No Content) - Normal success response (rather than 200 OK, since
200 OK would imply a response body, and an UNLOCK success response
does not normally contain a body).

400 (Bad Request) - No lock token was provided.

403 (Forbidden) - The currently authenticated principal does not have
permission to remove the lock.

409 (Conflict), with 'lock-token-matches-request-uri' precondition -
The resource was not locked, or the request was made to a Request-URI
that was not within the scope of the lock.

[9.11.2](#). Example - UNLOCK

   >>Request

     UNLOCK /workspace/webdav/info.doc HTTP/1.1
     Host: example.com
     Lock-Token: <urn:uuid:a515cfa4-5da4-22e1-f5b5-00a0451e6bf7>
     Authorization: Digest username="ejw"
       realm="ejw@example.com", nonce="...",
       uri="/workspace/webdav/proposal.doc",
       response="...", opaque="..."

   >>Response

     HTTP/1.1 204 No Content

   In this example, the lock identified by the lock token
   "urn:uuid:a515cfa4-5da4-22e1-f5b5-00a0451e6bf7" is successfully
   removed from the resource
   http://example.com/workspace/webdav/info.doc. If this lock included
   more than just one resource, the lock is removed from all resources
   included in the lock.

   In this example, the nonce, response, and opaque fields have not been
   calculated in the Authorization request header.

[10](#). HTTP Headers for Distributed Authoring

   All DAV headers follow the same basic formatting rules as HTTP
   headers. This includes rules like line continuation and how to
   combine (or separate) multiple instances of the same header using
   commas.

WebDAV adds two new conditional headers to the set defined in HTTP:
the If and Overwrite headers.

10.1. DAV Header

```
DAV               = "DAV" ":" #( compliance-class )
compliance-class = ( "1" | "2" | "3" | extend )
extend            = Coded-URL | token
                     ; token is defined in RFC 2616, Section 2.2
Coded-URL         = "<" absolute-URI ">"
                     ; No linear whitespace (LWS) allowed in Coded-URL
                     ; absolute-URI defined in RFC 3986, Section 4.3
```

This general-header appearing in the response indicates that the
resource supports the DAV schema and protocol as specified. All DAV-
compliant resources MUST return the DAV header with compliance-class
"1" on all OPTIONS responses. In cases where WebDAV is only
supported in part of the server namespace, an OPTIONS request to non-
WebDAV resources (including "/") SHOULD NOT advertise WebDAV support.

The value is a comma-separated list of all compliance class
identifiers that the resource supports. Class identifiers may be
Coded-URLs or tokens (as defined by [RFC2616]). Identifiers can
appear in any order. Identifiers that are standardized through the
IETF RFC process are tokens, but other identifiers SHOULD be Coded-
URLs to encourage uniqueness.

A resource must show class 1 compliance if it shows class 2 or 3
compliance. In general, support for one compliance class does not
entail support for any other, and in particular, support for
compliance class 3 does not require support for compliance class 2.
Please refer to Section 18 for more details on compliance classes
defined in this specification.

Note that many WebDAV servers do not advertise WebDAV support in
response to "OPTIONS *".

As a request header, this header allows the client to advertise

compliance with named features when the server needs that
information. Clients SHOULD NOT send this header unless a standards
track specification requires it. Any extension that makes use of
this as a request header will need to carefully consider caching
implications.

10.2. Depth Header

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

   The Depth request header is used with methods executed on resources
   that could potentially have internal members to indicate whether the
   method is to be applied only to the resource ("Depth: 0"), to the
   resource and its internal members only ("Depth: 1"), or the resource
   and all its members ("Depth: infinity").

   The Depth header is only supported if a method's definition
   explicitly provides for such support.

   The following rules are the default behavior for any method that
   supports the Depth header. A method may override these defaults by
   defining different behavior in its definition.

   Methods that support the Depth header may choose not to support all
   of the header's values and may define, on a case-by-case basis, the
   behavior of the method if a Depth header is not present. For
   example, the MOVE method only supports "Depth: infinity", and if a
   Depth header is not present, it will act as if a "Depth: infinity"
   header had been applied.

   Clients MUST NOT rely upon methods executing on members of their
   hierarchies in any particular order or on the execution being atomic
   unless the particular method explicitly provides such guarantees.

   Upon execution, a method with a Depth header will perform as much of
   its assigned task as possible and then return a response specifying
   what it was able to accomplish and what it failed to do.

   So, for example, an attempt to COPY a hierarchy may result in some of
   the members being copied and some not.

By default, the Depth header does not interact with other headers.
That is, each header on a request with a Depth header MUST be applied
only to the Request-URI if it applies to any resource, unless
specific Depth behavior is defined for that header.

If a source or destination resource within the scope of the Depth
header is locked in such a way as to prevent the successful execution
of the method, then the lock token for that resource MUST be
submitted with the request in the If request header.

The Depth header only specifies the behavior of the method with
regards to internal members. If a resource does not have internal
members, then the Depth header MUST be ignored.

10.3. Destination Header

The Destination request header specifies the URI that identifies a
destination resource for methods such as COPY and MOVE, which take
two URIs as parameters.

       Destination = "Destination" ":" Simple-ref

If the Destination value is an absolute-URI (Section 4.3 of
   [RFC3986]), it may name a different server (or different port or
scheme). If the source server cannot attempt a copy to the remote
server, it MUST fail the request. Note that copying and moving
resources to remote servers is not fully defined in this
specification (e.g., specific error conditions).

If the Destination value is too long or otherwise unacceptable, the
server SHOULD return 400 (Bad Request), ideally with helpful
information in an error body.

10.4. If Header

The If request header is intended to have similar functionality to
the If-Match header defined in Section 14.24 of [RFC2616]. However,
the If header handles any state token as well as ETags. A typical
example of a state token is a lock token, and lock tokens are the
only state tokens defined in this specification.

. Purpose

   The If header has two distinct purposes:

   o The first purpose is to make a request conditional by supplying a
      series of state lists with conditions that match tokens and ETags
      to a specific resource. If this header is evaluated and all state
      lists fail, then the request MUST fail with a 412 (Precondition
      Failed) status. On the other hand, the request can succeed only
      if one of the described state lists succeeds. The success
      criteria for state lists and matching functions are defined in
      Sections [10.4.3](#) and [10.4.4](#).

   o Additionally, the mere fact that a state token appears in an If
      header means that it has been "submitted" with the request. In
      general, this is used to indicate that the client has knowledge of
      that state token. The semantics for submitting a state token
      depend on its type (for lock tokens, please refer to [Section 6](#)).

   Note that these two purposes need to be treated distinctly: a state
   token counts as being submitted independently of whether the server
   actually has evaluated the state list it appears in, and also
   independently of whether or not the condition it expressed was found
   to be true.

[10.4.2](#). Syntax

      If = "If" ":" (1*No-tag-list | 1*Tagged-list)

      No-tag-list = List
      Tagged-list = Resource-Tag 1*List

      List = "(" 1*Condition ")"
      Condition = ["Not"] (State-token | "[" entity-tag "]")
      ; entity-tag: see [Section 3.11 of [RFC2616]](#)
      ; No LWS allowed between "[", entity-tag and "]"

      State-token = Coded-URL

      Resource-Tag = "<" Simple-ref ">"
      ; Simple-ref: see [Section 8.3](#)

```
   ; No LWS allowed in Resource-Tag
```

   The syntax distinguishes between untagged lists ("No-tag-list") and
   tagged lists ("Tagged-list"). Untagged lists apply to the resource
   identified by the Request-URI, while tagged lists apply to the
   resource identified by the preceding Resource-Tag.

   A Resource-Tag applies to all subsequent Lists, up to the next
   Resource-Tag.

   Note that the two list types cannot be mixed within an If header.
   This is not a functional restriction because the No-tag-list syntax
   is just a shorthand notation for a Tagged-list production with a
   Resource-Tag referring to the Request-URI.

   Each List consists of one or more Conditions. Each Condition is
   defined in terms of an entity-tag or state-token, potentially negated
   by the prefix "Not".

   Note that the If header syntax does not allow multiple instances of
   If headers in a single request. However, the HTTP header syntax
   allows extending single header values across multiple lines, by
   inserting a line break followed by whitespace (see [RFC2616], Section
   4.2).

10.4.3. List Evaluation

   A Condition that consists of a single entity-tag or state-token
   evaluates to true if the resource matches the described state (where
   the individual matching functions are defined below in
   Section 10.4.4). Prefixing it with "Not" reverses the result of the
   evaluation (thus, the "Not" applies only to the subsequent entity-tag
   or state-token).

   Each List production describes a series of conditions. The whole
   list evaluates to true if and only if each condition evaluates to
   true (that is, the list represents a logical conjunction of
   Conditions).

   Each No-tag-list and Tagged-list production may contain one or more
   Lists. They evaluate to true if and only if any of the contained
   lists evaluates to true (that is, if there's more than one List, that
   List sequence represents a logical disjunction of the Lists).

Finally, the whole If header evaluates to true if and only if at
least one of the No-tag-list or Tagged-list productions evaluates to
true. If the header evaluates to false, the server MUST reject the
request with a 412 (Precondition Failed) status. Otherwise,
execution of the request can proceed as if the header wasn't present.

10.4.4. Matching State Tokens and ETags

When performing If header processing, the definition of a matching
state token or entity tag is as follows:

Identifying a resource: The resource is identified by the URI along
with the token, in tagged list production, or by the Request-URI in
untagged list production.

Matching entity tag: Where the entity tag matches an entity tag
associated with the identified resource. Servers MUST use either the
weak or the strong comparison function defined in Section 13.3.3 of
   [RFC2616].

Matching state token: Where there is an exact match between the state
token in the If header and any state token on the identified
resource. A lock state token is considered to match if the resource
is anywhere in the scope of the lock.

Handling unmapped URLs: For both ETags and state tokens, treat as if
the URL identified a resource that exists but does not have the
specified state.

10.4.5. If Header and Non-DAV-Aware Proxies

Non-DAV-aware proxies will not honor the If header, since they will
not understand the If header, and HTTP requires non-understood
headers to be ignored. When communicating with HTTP/1.1 proxies, the
client MUST use the "Cache-Control: no-cache" request header so as to
prevent the proxy from improperly trying to service the request from
its cache. When dealing with HTTP/1.0 proxies, the "Pragma: no-
cache" request header MUST be used for the same reason.

Because in general clients may not be able to reliably detect non-
DAV-aware intermediates, they are advised to always prevent caching
using the request directives mentioned above.

10.4.6. Example - No-tag Production

```
If: (<urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2>
  ["I am an ETag"])
  (["I am another ETag"])
```

The previous header would require that the resource identified in the
Request-URI be locked with the specified lock token and be in the
state identified by the "I am an ETag" ETag or in the state
identified by the second ETag "I am another ETag".

To put the matter more plainly one can think of the previous If
header as expressing the condition below:

```
(
  is-locked-with(urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2) AND
  matches-etag("I am an ETag")
)
OR
(
  matches-etag("I am another ETag")
)
```

10.4.7. Example - Using "Not" with No-tag Production

```
If: (Not <urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2>
  <urn:uuid:58f202ac-22cf-11d1-b12d-002035b29092>)
```

This If header requires that the resource must not be locked with a
lock having the lock token
urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2 and must be locked by a
lock with the lock token
urn:uuid:58f202ac-22cf-11d1-b12d-002035b29092.

10.4.8. Example - Causing a Condition to Always Evaluate to True

There may be cases where a client wishes to submit state tokens, but
doesn't want the request to fail just because the state token isn't
current anymore. One simple way to do this is to include a Condition
that is known to always evaluate to true, such as in:

If: (<urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2>)
           (Not <DAV:no-lock>)

    "DAV:no-lock" is known to never represent a current lock token. Lock
    tokens are assigned by the server, following the uniqueness
    requirements described in Section 6.5, therefore cannot use the
    "DAV:" scheme. Thus, by applying "Not" to a state token that is

    known not to be current, the Condition always evaluates to true.
    Consequently, the whole If header will always evaluate to true, and
    the lock token urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2 will be
    submitted in any case.

10.4.9. Example - Tagged List If Header in COPY

    >>Request

      COPY /resource1 HTTP/1.1
      Host: www.example.com
      Destination: /resource2
      If: </resource1>
         (<urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2>
         [W/"A weak ETag"]) (["strong ETag"])

    In this example, http://www.example.com/resource1 is being copied to
    http://www.example.com/resource2. When the method is first applied
    to http://www.example.com/resource1, resource1 must be in the state
    specified by "(<urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2> [W/"A
    weak ETag"]) (["strong ETag"])". That is, either it must be locked
    with a lock token of "urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2"
    and have a weak entity tag W/"A weak ETag" or it must have a strong
    entity tag "strong ETag".

10.4.10. Example - Matching Lock Tokens with Collection Locks

      DELETE /specs/rfc2518.txt HTTP/1.1
      Host: www.example.com
      If: <http://www.example.com/specs/>
         (<urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2>)

    For this example, the lock token must be compared to the identified

resource, which is the 'specs' collection identified by the URL in
the tagged list production. If the 'specs' collection is not locked
by a lock with the specified lock token, the request MUST fail.
Otherwise, this request could succeed, because the If header
evaluates to true, and because the lock token for the lock affecting
the affected resource has been submitted.

10.4.11. Example - Matching ETags on Unmapped URLs

   Consider a collection "/specs" that does not contain the member
   "/specs/rfc2518.doc". In this case, the If header

      If: </specs/rfc2518.doc> (["4217"])

   will evaluate to false (the URI isn't mapped, thus the resource
   identified by the URI doesn't have an entity matching the ETag
   "4217").

   On the other hand, an If header of

      If: </specs/rfc2518.doc> (Not ["4217"])

   will consequently evaluate to true.

   Note that, as defined above in Section 10.4.4, the same
   considerations apply to matching state tokens.

10.5. Lock-Token Header

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

   The Lock-Token request header is used with the UNLOCK method to
   identify the lock to be removed. The lock token in the Lock-Token
   request header MUST identify a lock that contains the resource
   identified by Request-URI as a member.

   The Lock-Token response header is used with the LOCK method to
   indicate the lock token created as a result of a successful LOCK
   request to create a new lock.

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

   The Overwrite request header specifies whether the server should
   overwrite a resource mapped to the destination URL during a COPY or
   MOVE. A value of "F" states that the server must not perform the
   COPY or MOVE operation if the destination URL does map to a resource.
   If the overwrite header is not included in a COPY or MOVE request,
   then the resource MUST treat the request as if it has an overwrite
   header of value "T". While the Overwrite header appears to duplicate
   the functionality of using an "If-Match: *" header (see [[RFC2616]()]),
   If-Match applies only to the Request-URI, and not to the Destination
   of a COPY or MOVE.

   If a COPY or MOVE is not performed due to the value of the Overwrite
   header, the method MUST fail with a 412 (Precondition Failed) status
   code. The server MUST do authorization checks before checking this
   or any conditional header.

   All DAV-compliant resources MUST support the Overwrite header.

    TimeOut = "Timeout" ":" 1#TimeType
    TimeType = ("Second-" DAVTimeOutVal | "Infinite")
               ; No LWS allowed within TimeType
    DAVTimeOutVal = 1*DIGIT

   Clients MAY include Timeout request headers in their LOCK requests.
   However, the server is not required to honor or even consider these
   requests. Clients MUST NOT submit a Timeout request header with any
   method other than a LOCK method.

   The "Second" TimeType specifies the number of seconds that will
   elapse between granting of the lock at the server, and the automatic
   removal of the lock. The timeout value for TimeType "Second" MUST
   NOT be greater than $2^{32}-1$.

   See [Section 6.6]() for a description of lock timeout behavior.

11. Status Code Extensions to HTTP/1.1

The following status codes are added to those defined in HTTP/1.1
[RFC2616].

11.1. 207 Multi-Status

The 207 (Multi-Status) status code provides status for multiple
independent operations (see Section 13 for more information).

11.2. 422 Unprocessable Entity

The 422 (Unprocessable Entity) status code means the server
understands the content type of the request entity (hence a
415(Unsupported Media Type) status code is inappropriate), and the
syntax of the request entity is correct (thus a 400 (Bad Request)
status code is inappropriate) but was unable to process the contained
instructions. For example, this error condition may occur if an XML
request body contains well-formed (i.e., syntactically correct), but
semantically erroneous, XML instructions.

11.3. 423 Locked

The 423 (Locked) status code means the source or destination resource
of a method is locked. This response SHOULD contain an appropriate
precondition or postcondition code, such as 'lock-token-submitted' or
'no-conflicting-lock'.

11.4. 424 Failed Dependency

The 424 (Failed Dependency) status code means that the method could
not be performed on the resource because the requested action
depended on another action and that action failed. For example, if a
command in a PROPPATCH method fails, then, at minimum, the rest of
the commands will also fail with 424 (Failed Dependency).

11.5. 507 Insufficient Storage

The 507 (Insufficient Storage) status code means the method could not
be performed on the resource because the server is unable to store

the representation needed to successfully complete the request. This
condition is considered to be temporary. If the request that
received this status code was the result of a user action, the
request MUST NOT be repeated until it is requested by a separate user
action.

12. Use of HTTP Status Codes

These HTTP codes are not redefined, but their use is somewhat
extended by WebDAV methods and requirements. In general, many HTTP
status codes can be used in response to any request, not just in
cases described in this document. Note also that WebDAV servers are
known to use 300-level redirect responses (and early interoperability
tests found clients unprepared to see those responses). A 300-level
response MUST NOT be used when the server has created a new resource
in response to the request.

12.1. 412 Precondition Failed

Any request can contain a conditional header defined in HTTP (If-
Match, If-Modified-Since, etc.) or the "If" or "Overwrite"
conditional headers defined in this specification. If the server
evaluates a conditional header, and if that condition fails to hold,
then this error code MUST be returned. On the other hand, if the
client did not include a conditional header in the request, then the
server MUST NOT use this status code.

12.2. 414 Request-URI Too Long

This status code is used in HTTP 1.1 only for Request-URIs, not URIs
in other locations.

13. Multi-Status Response

A Multi-Status response conveys information about multiple resources
in situations where multiple status codes might be appropriate. The
default Multi-Status response body is a text/xml or application/xml

HTTP entity with a 'multistatus' root element. Further elements
contain 200, 300, 400, and 500 series status codes generated during
the method invocation. 100 series status codes SHOULD NOT be recorded
in a 'response' XML element.

Although '207' is used as the overall response status code, the
recipient needs to consult the contents of the multistatus response
body for further information about the success or failure of the
method execution. The response MAY be used in success, partial
success and also in failure situations.

The 'multistatus' root element holds zero or more 'response' elements
in any order, each with information about an individual resource.
Each 'response' element MUST have an 'href' element to identify the
resource.

A Multi-Status response uses one out of two distinct formats for
representing the status:

1. A 'status' element as child of the 'response' element indicates
    the status of the message execution for the identified resource
    as a whole (for instance, see Section 9.6.2). Some method
    definitions provide information about specific status codes
    clients should be prepared to see in a response. However,
    clients MUST be able to handle other status codes, using the
    generic rules defined in Section 10 of [RFC2616].

2. For PROPFIND and PROPPATCH, the format has been extended using
    the 'propstat' element instead of 'status', providing information
    about individual properties of a resource. This format is
    specific to PROPFIND and PROPPATCH, and is described in detail in
    Sections 9.1 and 9.2.

13.1. Response Headers

   HTTP defines the Location header to indicate a preferred URL for the
   resource that was addressed in the Request-URI (e.g., in response to
   successful PUT requests or in redirect responses). However, use of
   this header creates ambiguity when there are URLs in the body of the
   response, as with Multi-Status. Thus, use of the Location header
   with the Multi-Status response is intentionally undefined.

13.2. Handling Redirected Child Resources

Redirect responses (300-303, 305, and 307) defined in HTTP 1.1 normally take a Location header to indicate the new URI for the single resource redirected from the Request-URI. Multi-Status responses contain many resource addresses, but the original definition in [RFC2518] did not have any place for the server to provide the new URI for redirected resources. This specification does define a 'location' element for this information (see Section 14.9). Servers MUST use this new element with redirect responses in Multi-Status.

Clients encountering redirected resources in Multi-Status MUST NOT rely on the 'location' element being present with a new URI. If the element is not present, the client MAY reissue the request to the individual redirected resource, because the response to that request can be redirected with a Location header containing the new URI.

13.3. Internal Status Codes

Sections 9.2.1, 9.1.2, 9.6.1, 9.8.3, and 9.9.2 define various status codes used in Multi-Status responses. This specification does not define the meaning of other status codes that could appear in these responses.

14. XML Element Definitions

In this section, the final line of each section gives the element type declaration using the format defined in [REC-XML]. The "Value" field, where present, specifies further restrictions on the allowable contents of the XML element using BNF (i.e., to further restrict the values of a PCDATA element). Note that all of the elements defined here may be extended according to the rules defined in Section 17. All elements defined here are in the "DAV:" namespace.

14.1. activelock XML Element

Name: activelock

Purpose: Describes a lock on a resource.


```
<!ELEMENT activelock (lockscope, locktype, depth, owner?, timeout?,
          locktoken?, lockroot)>
```

[14.2](#). allprop XML Element

   Name: allprop

   Purpose: Specifies that all names and values of dead properties and
      the live properties defined by this document existing on the
      resource are to be returned.

   <!ELEMENT allprop EMPTY >

[14.3](#). collection XML Element

   Name: collection

   Purpose: Identifies the associated resource as a collection. The
      DAV:resourcetype property of a collection resource MUST contain
      this element. It is normally empty but extensions may add sub-
      elements.

   <!ELEMENT collection EMPTY >

[14.4](#). depth XML Element

   Name: depth

   Purpose: Used for representing depth values in XML content (e.g.,
      in lock information).

   Value: "0" | "1" | "infinity"

   <!ELEMENT depth (#PCDATA) >

[14.5](#). error XML Element

   Name: error

   Purpose: Error responses, particularly 403 Forbidden and 409
      Conflict, sometimes need more information to indicate what went
      wrong. In these cases, servers MAY return an XML response body
      with a document element of 'error', containing child elements
      identifying particular condition codes.

Description: Contains at least one XML element, and MUST NOT
    contain text or mixed content. Any element that is a child of the
    'error' element is considered to be a precondition or
    postcondition code. Unrecognized elements MUST be ignored.

    <!ELEMENT error ANY >

[14.6](#).   exclusive XML Element

    Name: exclusive

    Purpose: Specifies an exclusive lock.

    <!ELEMENT exclusive EMPTY >

[14.7](#).   href XML Element

    Name: href

    Purpose: MUST contain a URI or a relative reference.

    Description: There may be limits on the value of 'href' depending
        on the context of its use. Refer to the specification text where
        'href' is used to see what limitations apply in each case.

    Value: Simple-ref

    <!ELEMENT href (#PCDATA)>

[14.8](#).   include XML Element

    Name: include

    Purpose: Any child element represents the name of a property to be
        included in the PROPFIND response. All elements inside an
        'include' XML element MUST define properties related to the
        resource, although possible property names are in no way limited
        to those property names defined in this document or other
        standards. This element MUST NOT contain text or mixed content.

```
<!ELEMENT include ANY >
```

14.9. location XML Element

   Name: location

   Purpose: HTTP defines the "Location" header (see [RFC2616], Section
      14.30) for use with some status codes (such as 201 and the 300
      series codes). When these codes are used inside a 'multistatus'
      element, the 'location' element can be used to provide the
      accompanying Location header value.

   Description: Contains a single href element with the same value
      that would be used in a Location header.

```
<!ELEMENT location (href)>
```

14.10. lockentry XML Element

   Name: lockentry

   Purpose: Defines the types of locks that can be used with the
      resource.

```
<!ELEMENT lockentry (lockscope, locktype) >
```

14.11. lockinfo XML Element

   Name: lockinfo

   Purpose: The 'lockinfo' XML element is used with a LOCK method to
      specify the type of lock the client wishes to have created.

```
<!ELEMENT lockinfo (lockscope, locktype, owner?)  >
```

14.12. lockroot XML Element

   Name: lockroot

Purpose: Contains the root URL of the lock, which is the URL
      through which the resource was addressed in the LOCK request.

   Description: The href element contains the root of the lock. The
      server SHOULD include this in all DAV:lockdiscovery property
      values and the response to LOCK requests.

   <!ELEMENT lockroot (href) >

14.13. lockscope XML Element

   Name: lockscope

   Purpose: Specifies whether a lock is an exclusive lock, or a shared
      lock.

      <!ELEMENT lockscope (exclusive | shared) >

14.14. locktoken XML Element

   Name: locktoken

   Purpose: The lock token associated with a lock.

   Description: The href contains a single lock token URI, which
      refers to the lock.

   <!ELEMENT locktoken (href) >

14.15. locktype XML Element

   Name: locktype

   Purpose: Specifies the access type of a lock. At present, this
      specification only defines one lock type, the write lock.

   <!ELEMENT locktype (write) >

. multistatus XML Element

   Name: multistatus

   Purpose: Contains multiple response messages.

   Description: The 'responsedescription' element at the top level is
      used to provide a general message describing the overarching
      nature of the response. If this value is available, an
      application may use it instead of presenting the individual
      response descriptions contained within the responses.

   <!ELEMENT multistatus (response*, responsedescription?) >

[14.17](). owner XML Element

   Name: owner

   Purpose: Holds client-supplied information about the creator of a
      lock.

   Description: Allows a client to provide information sufficient for
      either directly contacting a principal (such as a telephone number
      or Email URI), or for discovering the principal (such as the URL

Dusseault   Standards Track   [Page 85]

[RFC 4918]()   WebDAV   June 2007

      of a homepage) who created a lock. The value provided MUST be
      treated as a dead property in terms of XML Information Item
      preservation. The server MUST NOT alter the value unless the
      owner value provided by the client is empty. For a certain amount
      of interoperability between different client implementations, if
      clients have URI-formatted contact information for the lock
      creator suitable for user display, then clients SHOULD put those
      URIs in 'href' child elements of the 'owner' element.

   Extensibility: MAY be extended with child elements, mixed content,
      text content or attributes.

   <!ELEMENT owner ANY >

14.18. prop XML Element

   Name: prop

   Purpose: Contains properties related to a resource.

   Description: A generic container for properties defined on
      resources. All elements inside a 'prop' XML element MUST define
      properties related to the resource, although possible property
      names are in no way limited to those property names defined in
      this document or other standards. This element MUST NOT contain
      text or mixed content.

   <!ELEMENT prop ANY >

14.19. propertyupdate XML Element

   Name: propertyupdate

   Purpose: Contains a request to alter the properties on a resource.

   Description: This XML element is a container for the information
      required to modify the properties on the resource.

   <!ELEMENT propertyupdate (remove | set)+ >

14.20. propfind XML Element

   Name: propfind

   Purpose: Specifies the properties to be returned from a PROPFIND
      method. Four special elements are specified for use with
      'propfind': 'prop', 'allprop', 'include', and 'propname'. If
      'prop' is used inside 'propfind', it MUST NOT contain property
      values.

   <!ELEMENT propfind (propname | (allprop, include?) | prop) >

14.21. propname XML Element

   Name: propname

   Purpose: Specifies that only a list of property names on the
      resource is to be returned.

   <!ELEMENT propname EMPTY >

14.22. propstat XML Element

   Name: propstat

   Purpose: Groups together a prop and status element that is
      associated with a particular 'href' element.

   Description: The propstat XML element MUST contain one prop XML
      element and one status XML element. The contents of the prop XML
      element MUST only list the names of properties to which the result
      in the status element applies. The optional precondition/
      postcondition element and 'responsedescription' text also apply to
      the properties named in 'prop'.

   <!ELEMENT propstat (prop, status, error?, responsedescription?) >

14.23. remove XML Element

   Name: remove

   Purpose: Lists the properties to be removed from a resource.

   Description: Remove instructs that the properties specified in prop
      should be removed. Specifying the removal of a property that does
      not exist is not an error. All the XML elements in a 'prop' XML
      element inside of a 'remove' XML element MUST be empty, as only
      the names of properties to be removed are required.

   <!ELEMENT remove (prop) >

.   response XML Element

   Name: response

   Purpose: Holds a single response describing the effect of a method
      on resource and/or its properties.

   Description: The 'href' element contains an HTTP URL pointing to a
      WebDAV resource when used in the 'response' container. A
      particular 'href' value MUST NOT appear more than once as the
      child of a 'response' XML element under a 'multistatus' XML
      element. This requirement is necessary in order to keep
      processing costs for a response to linear time. Essentially, this
      prevents having to search in order to group together all the
      responses by 'href'. There are, however, no requirements
      regarding ordering based on 'href' values. The optional
      precondition/postcondition element and 'responsedescription' text
      can provide additional information about this resource relative to
      the request or result.

      <!ELEMENT response (href, ((href*, status)|(propstat+)),
                        error?, responsedescription? , location?) >

14.25.   responsedescription XML Element

   Name: responsedescription

   Purpose: Contains information about a status response within a
      Multi-Status.

   Description: Provides information suitable to be presented to a
      user.

   <!ELEMENT responsedescription (#PCDATA) >

14.26.   set XML Element

   Name: set

   Purpose: Lists the property values to be set for a resource.

   Description: The 'set' element MUST contain only a 'prop' element.
      The elements contained by the 'prop' element inside the 'set'
      element MUST specify the name and value of properties that are set
      on the resource identified by Request-URI. If a property already
      exists, then its value is replaced. Language tagging information
      appearing in the scope of the 'prop' element (in the "xml:lang"

attribute, if present) MUST be persistently stored along with the
property, and MUST be subsequently retrievable using PROPFIND.

```
<!ELEMENT set (prop) >
```

14.27. shared XML Element

Name: shared

Purpose: Specifies a shared lock.

```
<!ELEMENT shared EMPTY >
```

14.28. status XML Element

Name: status

Purpose: Holds a single HTTP status-line.

Value: status-line (defined in Section 6.1 of [RFC2616])

```
<!ELEMENT status (#PCDATA) >
```

14.29. timeout XML Element

Name: timeout

Purpose: The number of seconds remaining before a lock expires.

Value: TimeType (defined in Section 10.7)

```
<!ELEMENT timeout (#PCDATA) >
```

14.30. write XML Element

Name: write

Purpose: Specifies a write lock.

```
<!ELEMENT write EMPTY >
```

15. DAV Properties

For DAV properties, the name of the property is also the same as the
name of the XML element that contains its value. In the section
below, the final line of each section gives the element type
declaration using the format defined in [[REC-XML](#)]. The "Value"
field, where present, specifies further restrictions on the allowable
contents of the XML element using BNF (i.e., to further restrict the
values of a PCDATA element).

A protected property is one that cannot be changed with a PROPPATCH
request. There may be other requests that would result in a change
to a protected property (as when a LOCK request affects the value of
DAV:lockdiscovery). Note that a given property could be protected on
one type of resource, but not protected on another type of resource.

A computed property is one with a value defined in terms of a
computation (based on the content and other properties of that
resource, or even of some other resource). A computed property is
always a protected property.

COPY and MOVE behavior refers to local COPY and MOVE operations.

For properties defined based on HTTP GET response headers (DAV:get*),
the header value could include LWS as defined in [[RFC2616](#)], [Section
4.2](#). Server implementors SHOULD strip LWS from these values before
using as WebDAV property values.

15.1. creationdate Property

Name: creationdate

Purpose: Records the time and date the resource was created.

Value: date-time (defined in [[RFC3339](#)], see the ABNF in [Section
   5.6](#).)

Protected: MAY be protected. Some servers allow DAV:creationdate
      to be changed to reflect the time the document was created if that
      is more meaningful to the user (rather than the time it was
      uploaded). Thus, clients SHOULD NOT use this property in
      synchronization logic (use DAV:getetag instead).

   COPY/MOVE behavior: This property value SHOULD be kept during a
      MOVE operation, but is normally re-initialized when a resource is
      created with a COPY. It should not be set in a COPY.

   Description: The DAV:creationdate property SHOULD be defined on all
      DAV compliant resources. If present, it contains a timestamp of
      the moment when the resource was created. Servers that are
      incapable of persistently recording the creation date SHOULD
      instead leave it undefined (i.e. report "Not Found").

   <!ELEMENT creationdate (#PCDATA) >

15.2. displayname Property

   Name: displayname

   Purpose: Provides a name for the resource that is suitable for
      presentation to a user.

   Value: Any text.

   Protected: SHOULD NOT be protected. Note that servers implementing
      [RFC2518] might have made this a protected property as this is a
      new requirement.

   COPY/MOVE behavior: This property value SHOULD be preserved in COPY
      and MOVE operations.

   Description: Contains a description of the resource that is
      suitable for presentation to a user. This property is defined on
      the resource, and hence SHOULD have the same value independent of
      the Request-URI used to retrieve it (thus, computing this property
      based on the Request-URI is deprecated). While generic clients

might display the property value to end users, client UI designers must understand that the method for identifying resources is still the URL. Changes to DAV:displayname do not issue moves or copies to the server, but simply change a piece of meta-data on the individual resource. Two resources can have the same DAV: displayname value even within the same collection.

```
<!ELEMENT displayname (#PCDATA) >
```

15.3. getcontentlanguage Property

Name: getcontentlanguage

Purpose: Contains the Content-Language header value (from Section 14.12 of [RFC2616]) as it would be returned by a GET without accept headers.

Value: language-tag (language-tag is defined in Section 3.10 of [RFC2616])

Protected: SHOULD NOT be protected, so that clients can reset the language. Note that servers implementing [RFC2518] might have made this a protected property as this is a new requirement.

COPY/MOVE behavior: This property value SHOULD be preserved in COPY and MOVE operations.

Description: The DAV:getcontentlanguage property MUST be defined on any DAV-compliant resource that returns the Content-Language header on a GET.

```
<!ELEMENT getcontentlanguage (#PCDATA) >
```

15.4. getcontentlength Property

Name: getcontentlength

Purpose: Contains the Content-Length header returned by a GET without accept headers.

Value: See Section 14.13 of [RFC2616].

Protected: This property is computed, therefore protected.

   Description: The DAV:getcontentlength property MUST be defined on
      any DAV-compliant resource that returns the Content-Length header
      in response to a GET.

   COPY/MOVE behavior: This property value is dependent on the size of
      the destination resource, not the value of the property on the
      source resource.

   <!ELEMENT getcontentlength (#PCDATA) >

15.5. getcontenttype Property

   Name: getcontenttype

   Purpose: Contains the Content-Type header value (from Section 14.17
      of [RFC2616]) as it would be returned by a GET without accept
      headers.

   Value: media-type (defined in Section 3.7 of [RFC2616])

   Protected: Potentially protected if the server prefers to assign
      content types on its own (see also discussion in Section 9.7.1).

   COPY/MOVE behavior: This property value SHOULD be preserved in COPY
      and MOVE operations.

   Description: This property MUST be defined on any DAV-compliant
      resource that returns the Content-Type header in response to a
      GET.

   <!ELEMENT getcontenttype (#PCDATA) >

15.6. getetag Property

   Name: getetag

   Purpose: Contains the ETag header value (from Section 14.19 of
      [RFC2616]) as it would be returned by a GET without accept

headers.

Value: entity-tag (defined in [Section 3.11 of [RFC2616]](#))

Protected: MUST be protected because this value is created and
controlled by the server.

COPY/MOVE behavior: This property value is dependent on the final
state of the destination resource, not the value of the property
on the source resource. Also note the considerations in
[Section 8.8](#).

Description: The getetag property MUST be defined on any DAV-
compliant resource that returns the Etag header. Refer to [Section
3.11 of RFC 2616](#) for a complete definition of the semantics of an
ETag, and to [Section 8.6](#) for a discussion of ETags in WebDAV.

    <!ELEMENT getetag (#PCDATA) >

15.7. getlastmodified Property

Name: getlastmodified

Purpose: Contains the Last-Modified header value (from [Section
14.29 of [RFC2616]](#)) as it would be returned by a GET method
without accept headers.

Value: [rfc1123](#)-date (defined in [Section 3.3.1 of [RFC2616]](#))

Protected: SHOULD be protected because some clients may rely on the
value for appropriate caching behavior, or on the value of the
Last-Modified header to which this property is linked.

COPY/MOVE behavior: This property value is dependent on the last
modified date of the destination resource, not the value of the
property on the source resource. Note that some server
implementations use the file system date modified value for the
DAV:getlastmodified value, and this can be preserved in a MOVE
even when the HTTP Last-Modified value SHOULD change. Note that
since [[RFC2616](#)] requires clients to use ETags where provided, a
server implementing ETags can count on clients using a much better

mechanism than modification dates for offline synchronization or
cache control. Also note the considerations in [Section 8.8](#).

Description: The last-modified date on a resource SHOULD only
reflect changes in the body (the GET responses) of the resource.
A change in a property only SHOULD NOT cause the last-modified
date to change, because clients MAY rely on the last-modified date
to know when to overwrite the existing body. The DAV:
getlastmodified property MUST be defined on any DAV-compliant
resource that returns the Last-Modified header in response to a
GET.

    <!ELEMENT getlastmodified (#PCDATA) >

[15.8](#). lockdiscovery Property

Name: lockdiscovery

Purpose: Describes the active locks on a resource

Protected: MUST be protected. Clients change the list of locks
through LOCK and UNLOCK, not through PROPPATCH.

COPY/MOVE behavior: The value of this property depends on the lock
state of the destination, not on the locks of the source resource.
Recall that locks are not moved in a MOVE operation.

Description: Returns a listing of who has a lock, what type of lock
he has, the timeout type and the time remaining on the timeout,
and the associated lock token. Owner information MAY be omitted
if it is considered sensitive. If there are no locks, but the
server supports locks, the property will be present but contain
zero 'activelock' elements. If there are one or more locks, an
'activelock' element appears for each lock on the resource. This
property is NOT lockable with respect to write locks ([Section 7](#)).

    <!ELEMENT lockdiscovery (activelock)* >

[15.8.1](#). Example - Retrieving DAV:lockdiscovery

```
   >>Request

     PROPFIND /container/ HTTP/1.1
     Host: www.example.com
     Content-Length: xxxx
     Content-Type: application/xml; charset="utf-8"

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

   >>Response

     HTTP/1.1 207 Multi-Status
     Content-Type: application/xml; charset="utf-8"
     Content-Length: xxxx

     <?xml version="1.0" encoding="utf-8" ?>
     <D:multistatus xmlns:D='DAV:'>
       <D:response>
         <D:href>http://www.example.com/container/</D:href>
         <D:propstat>
           <D:prop>
             <D:lockdiscovery>
              <D:activelock>
                <D:locktype><D:write/></D:locktype>
                <D:lockscope><D:exclusive/></D:lockscope>
                <D:depth>0</D:depth>
                <D:owner>Jane Smith</D:owner>
                <D:timeout>Infinite</D:timeout>
                <D:locktoken>
                   <D:href
              >urn:uuid:f81de2ad-7f3d-a1b2-4f3c-00a0c91a9d76</D:href>
                </D:locktoken>
                <D:lockroot>
                   <D:href>http://www.example.com/container/</D:href>
                </D:lockroot>
              </D:activelock>
             </D:lockdiscovery>
           </D:prop>
           <D:status>HTTP/1.1 200 OK</D:status>
         </D:propstat>
       </D:response>
     </D:multistatus>
```

This resource has a single exclusive write lock on it, with an
infinite timeout.

15.9. resourcetype Property

   Name: resourcetype

   Purpose: Specifies the nature of the resource.

   Protected: SHOULD be protected. Resource type is generally decided
      through the operation creating the resource (MKCOL vs PUT), not by
      PROPPATCH.

   COPY/MOVE behavior: Generally a COPY/MOVE of a resource results in
      the same type of resource at the destination.

   Description: MUST be defined on all DAV-compliant resources. Each
      child element identifies a specific type the resource belongs to,
      such as 'collection', which is the only resource type defined by
      this specification (see Section 14.3). If the element contains
      the 'collection' child element plus additional unrecognized
      elements, it should generally be treated as a collection. If the
      element contains no recognized child elements, it should be
      treated as a non-collection resource. The default value is empty.
      This element MUST NOT contain text or mixed content. Any custom
      child element is considered to be an identifier for a resource
      type.

   Example: (fictional example to show extensibility)

       <x:resourcetype xmlns:x="DAV:">
          <x:collection/>
          <f:search-results xmlns:f="http://www.example.com/ns"/>
       </x:resourcetype>

15.10. supportedlock Property

   Name: supportedlock

   Purpose: To provide a listing of the lock capabilities supported by
      the resource.

   Protected: MUST be protected. Servers, not clients, determine what
      lock mechanisms are supported.

   COPY/MOVE behavior: This property value is dependent on the kind of
      locks supported at the destination, not on the value of the
      property at the source resource. Servers attempting to COPY to a
      destination should not attempt to set this property at the
      destination.

   Description: Returns a listing of the combinations of scope and
      access types that may be specified in a lock request on the
      resource. Note that the actual contents are themselves controlled
      by access controls, so a server is not required to provide
      information the client is not authorized to see. This property is
      NOT lockable with respect to write locks ([Section 7](#)).

   <!ELEMENT supportedlock (lockentry)* >

[15.10.1](#). Example - Retrieving DAV:supportedlock

   >>Request

     PROPFIND /container/ HTTP/1.1
     Host: www.example.com
     Content-Length: xxxx
     Content-Type: application/xml; charset="utf-8"

     <?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
     Content-Type: application/xml; charset="utf-8"
     Content-Length: xxxx

     <?xml version="1.0" encoding="utf-8" ?>
     <D:multistatus xmlns:D="DAV:">
       <D:response>
         <D:href>http://www.example.com/container/</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>
```

[16](#). Precondition/Postcondition XML Elements

   As introduced in [Section 8.7](#), extra information on error conditions
   can be included in the body of many status responses. This section
   makes requirements on the use of the error body mechanism and
   introduces a number of precondition and postcondition codes.

   A "precondition" of a method describes the state of the server that
   must be true for that method to be performed. A "postcondition" of a
   method describes the state of the server that must be true after that
   method has been completed.

   Each precondition and postcondition has a unique XML element
   associated with it. In a 207 Multi-Status response, the XML element
   MUST appear inside an 'error' element in the appropriate 'propstat or
   'response' element depending on whether the condition applies to one
   or more properties or to the resource as a whole. In all other error
   responses where this specification's 'error' body is used, the
   precondition/postcondition XML element MUST be returned as the child
   of a top-level 'error' element in the response body, unless otherwise
   negotiated by the request, along with an appropriate response status.
   The most common response status codes are 403 (Forbidden) if the
   request should not be repeated because it will always fail, and 409

(Conflict) if it is expected that the user might be able to resolve
the conflict and resubmit the request. The 'error' element MAY
contain child elements with specific error information and MAY be
extended with any custom child elements.

This mechanism does not take the place of using a correct numeric
status code as defined here or in HTTP, because the client must
always be able to take a reasonable course of action based only on
the numeric code. However, it does remove the need to define new
numeric codes. The new machine-readable codes used for this purpose
are XML elements classified as preconditions and postconditions, so
naturally, any group defining a new condition code can use their own
namespace. As always, the "DAV:" namespace is reserved for use by
IETF-chartered WebDAV working groups.

   A server supporting this specification SHOULD use the XML error
   whenever a precondition or postcondition defined in this document is
   violated. For error conditions not specified in this document, the
   server MAY simply choose an appropriate numeric status and leave the
   response body blank. However, a server MAY instead use a custom
   condition code and other supporting text, because even when clients
   do not automatically recognize condition codes, they can be quite
   useful in interoperability testing and debugging.

   Example - Response with precondition code

   >>Response

      HTTP/1.1 423 Locked
      Content-Type: application/xml; charset="utf-8"
      Content-Length: xxxx

      <?xml version="1.0" encoding="utf-8" ?>
      <D:error xmlns:D="DAV:">
        <D:lock-token-submitted>
          <D:href>/workspace/webdav/</D:href>
        </D:lock-token-submitted>
      </D:error>

   In this example, a client unaware of a depth-infinity lock on the

parent collection "/workspace/webdav/" attempted to modify the
collection member "/workspace/webdav/proposal.doc".

Some other useful preconditions and postconditions have been defined
in other specifications extending WebDAV, such as [RFC3744] (see
particularly Section 7.1.1), [RFC3253], and [RFC3648].

All these elements are in the "DAV:" namespace. If not specified
otherwise, the content for each condition's XML element is defined to
be empty.

   Name: lock-token-matches-request-uri

   Use with: 409 Conflict

   Purpose: (precondition) -- A request may include a Lock-Token header
      to identify a lock for the UNLOCK method. However, if the
      Request-URI does not fall within the scope of the lock identified
      by the token, the server SHOULD use this error. The lock may have
      a scope that does not include the Request-URI, or the lock could
      have disappeared, or the token may be invalid.

   Name: lock-token-submitted (precondition)

   Use with: 423 Locked

   Purpose: The request could not succeed because a lock token should
      have been submitted. This element, if present, MUST contain at
      least one URL of a locked resource that prevented the request. In
      cases of MOVE, COPY, and DELETE where collection locks are
      involved, it can be difficult for the client to find out which
      locked resource made the request fail -- but the server is only
      responsible for returning one such locked resource. The server
      MAY return every locked resource that prevented the request from
      succeeding if it knows them all.

   <!ELEMENT lock-token-submitted (href+) >

   Name: no-conflicting-lock (precondition)

Use with: Typically 423 Locked

Purpose: A LOCK request failed due the presence of an already
   existing conflicting lock. Note that a lock can be in conflict
   although the resource to which the request was directed is only
   indirectly locked. In this case, the precondition code can be
   used to inform the client about the resource that is the root of
   the conflicting lock, avoiding a separate lookup of the
   "lockdiscovery" property.

   <!ELEMENT no-conflicting-lock (href)* >

   Name: no-external-entities

   Use with: 403 Forbidden

   Purpose: (precondition) -- If the server rejects a client request
      because the request body contains an external entity, the server
      SHOULD use this error.

   Name: preserved-live-properties

   Use with: 409 Conflict

   Purpose: (postcondition) -- The server received an otherwise-valid
      MOVE or COPY request, but cannot maintain the live properties with
      the same behavior at the destination. It may be that the server

      only supports some live properties in some parts of the
      repository, or simply has an internal error.

   Name: propfind-finite-depth

   Use with: 403 Forbidden

   Purpose: (precondition) -- This server does not allow infinite-depth
      PROPFIND requests on collections.

Name: cannot-modify-protected-property

     Use with: 403 Forbidden

     Purpose: (precondition) -- The client attempted to set a protected
        property in a PROPPATCH (such as DAV:getetag). See also
        [RFC3253], Section 3.12.

17. XML Extensibility in DAV

     The XML namespace extension ([REC-XML-NAMES]) is used in this
     specification in order to allow for new XML elements to be added
     without fear of colliding with other element names. Although WebDAV
     request and response bodies can be extended by arbitrary XML
     elements, which can be ignored by the message recipient, an XML
     element in the "DAV:" namespace SHOULD NOT be used in the request or
     response body unless that XML element is explicitly defined in an
     IETF RFC reviewed by a WebDAV working group.

     For WebDAV to be both extensible and backwards-compatible, both
     clients and servers need to know how to behave when unexpected or
     unrecognized command extensions are received. For XML processing,
     this means that clients and servers MUST process received XML
     documents as if unexpected elements and attributes (and all children
     of unrecognized elements) were not there. An unexpected element or
     attribute includes one that may be used in another context but is not
     expected here. Ignoring such items for purposes of processing can of
     course be consistent with logging all information or presenting for
     debugging.

     This restriction also applies to the processing, by clients, of DAV
     property values where unexpected XML elements SHOULD be ignored
     unless the property's schema declares otherwise.

     This restriction does not apply to setting dead DAV properties on the
     server where the server MUST record all XML elements.

     Additionally, this restriction does not apply to the use of XML where
     XML happens to be the content type of the entity body, for example,
     when used as the body of a PUT.

Processing instructions in XML SHOULD be ignored by recipients.
Thus, specifications extending WebDAV SHOULD NOT use processing
instructions to define normative behavior.

XML DTD fragments are included for all the XML elements defined in
this specification. However, correct XML will not be valid according
to any DTD due to namespace usage and extension rules. In
particular:

o Elements (from this specification) are in the "DAV:" namespace,

o Element ordering is irrelevant unless otherwise stated,

o Extension attributes MAY be added,

o For element type definitions of "ANY", the normative text
   definition for that element defines what can be in it and what
   that means.

o For element type definitions of "#PCDATA", extension elements MUST
   NOT be added.

o For other element type definitions, including "EMPTY", extension
   elements MAY be added.

Note that this means that elements containing elements cannot be
extended to contain text, and vice versa.

With DTD validation relaxed by the rules above, the constraints
described by the DTD fragments are normative (see for example
Appendix A). A recipient of a WebDAV message with an XML body MUST
NOT validate the XML document according to any hard-coded or
dynamically-declared DTD.

Note that this section describes backwards-compatible extensibility
rules. There might also be times when an extension is designed not
to be backwards-compatible, for example, defining an extension that
reuses an XML element defined in this document but omitting one of
the child elements required by the DTDs in this specification.

18. DAV Compliance Classes

   A DAV-compliant resource can advertise several classes of compliance.
   A client can discover the compliance classes of a resource by
   executing OPTIONS on the resource and examining the "DAV" header
   which is returned. Note particularly that resources, rather than
   servers, are spoken of as being compliant. That is because
   theoretically some resources on a server could support different
   feature sets. For example, a server could have a sub-repository
   where an advanced feature like versioning was supported, even if that
   feature was not supported on all sub-repositories.

   Since this document describes extensions to the HTTP/1.1 protocol,
   minimally all DAV-compliant resources, clients, and proxies MUST be
   compliant with [RFC2616].

   A resource that is class 2 or class 3 compliant must also be class 1
   compliant.

18.1. Class 1

   A class 1 compliant resource MUST meet all "MUST" requirements in all
   sections of this document.

   Class 1 compliant resources MUST return, at minimum, the value "1" in
   the DAV header on all responses to the OPTIONS method.

18.2. Class 2

   A class 2 compliant resource MUST meet all class 1 requirements and
   support the LOCK method, the DAV:supportedlock property, the DAV:
   lockdiscovery property, the Time-Out response header and the Lock-
   Token request header. A class 2 compliant resource SHOULD also
   support the Timeout request header and the 'owner' XML element.

   Class 2 compliant resources MUST return, at minimum, the values "1"
   and "2" in the DAV header on all responses to the OPTIONS method.

18.3. Class 3

   A resource can explicitly advertise its support for the revisions to
   [RFC2518] made in this document. Class 1 MUST be supported as well.
   Class 2 MAY be supported. Advertising class 3 support in addition to
   class 1 and 2 means that the server supports all the requirements in
   this specification. Advertising class 3 and class 1 support, but not
   class 2, means that the server supports all the requirements in this
   specification except possibly those that involve locking support.

       Example:

               DAV: 1, 3

[19](#). Internationalization Considerations

       In the realm of internationalization, this specification complies
       with the IETF Character Set Policy [[RFC2277](#)]. In this specification,
       human-readable fields can be found either in the value of a property,
       or in an error message returned in a response entity body. In both
       cases, the human-readable content is encoded using XML, which has
       explicit provisions for character set tagging and encoding, and
       requires that XML processors read XML elements encoded, at minimum,
       using the UTF-8 [[RFC3629](#)] and UTF-16 [[RFC2781](#)] encodings of the ISO
       10646 multilingual plane. XML examples in this specification
       demonstrate use of the charset parameter of the Content-Type header
       (defined in [[RFC3023](#)]), as well as XML charset declarations.

       XML also provides a language tagging capability for specifying the
       language of the contents of a particular XML element. The "xml:lang"
       attribute appears on an XML element to identify the language of its
       content and attributes. See [[REC-XML](#)] for definitions of values and
       scoping.

       WebDAV applications MUST support the character set tagging, character
       set encoding, and the language tagging functionality of the XML
       specification. Implementors of WebDAV applications are strongly
       encouraged to read "XML Media Types" [[RFC3023](#)] for instruction on
       which MIME media type to use for XML transport, and on use of the
       charset parameter of the Content-Type header.

       Names used within this specification fall into four categories: names
       of protocol elements such as methods and headers, names of XML
       elements, names of properties, and names of conditions. Naming of
       protocol elements follows the precedent of HTTP, using English names
       encoded in US-ASCII for methods and headers. Since these protocol
       elements are not visible to users, and are simply long token
       identifiers, they do not need to support multiple languages.
       Similarly, the names of XML elements used in this specification are
       not visible to the user and hence do not need to support multiple

languages.

WebDAV property names are qualified XML names (pairs of XML namespace
name and local name). Although some applications (e.g., a generic
property viewer) will display property names directly to their users,
it is expected that the typical application will use a fixed set of
properties, and will provide a mapping from the property name and
namespace to a human-readable field when displaying the property name

to a user. It is only in the case where the set of properties is not
known ahead of time that an application need display a property name
to a user. We recommend that applications provide human-readable
property names wherever feasible.

For error reporting, we follow the convention of HTTP/1.1 status
codes, including with each status code a short, English description
of the code (e.g., 423 (Locked)). While the possibility exists that
a poorly crafted user agent would display this message to a user,
internationalized applications will ignore this message, and display
an appropriate message in the user's language and character set.

Since interoperation of clients and servers does not require locale
information, this specification does not specify any mechanism for
transmission of this information.

20. Security Considerations

This section is provided to detail issues concerning security
implications of which WebDAV applications need to be aware.

All of the security considerations of HTTP/1.1 (discussed in
[RFC2616]) and XML (discussed in [RFC3023]) also apply to WebDAV. In
addition, the security risks inherent in remote authoring require
stronger authentication technology, introduce several new privacy
concerns, and may increase the hazards from poor server design.
These issues are detailed below.

20.1. Authentication of Clients

Due to their emphasis on authoring, WebDAV servers need to use
authentication technology to protect not just access to a network
resource, but the integrity of the resource as well. Furthermore,

the introduction of locking functionality requires support for
authentication.

A password sent in the clear over an insecure channel is an
inadequate means for protecting the accessibility and integrity of a
resource as the password may be intercepted. Since Basic
authentication for HTTP/1.1 performs essentially clear text
transmission of a password, Basic authentication MUST NOT be used to
authenticate a WebDAV client to a server unless the connection is
secure. Furthermore, a WebDAV server MUST NOT send a Basic
authentication challenge in a WWW-Authenticate header unless the
connection is secure. An example of a secure connection would be a
Transport Layer Security (TLS) connection employing a strong cipher
suite and server authentication.

   WebDAV applications MUST support the Digest authentication scheme
   [RFC2617]. Since Digest authentication verifies that both parties to
   a communication know a shared secret, a password, without having to
   send that secret in the clear, Digest authentication avoids the
   security problems inherent in Basic authentication while providing a
   level of authentication that is useful in a wide range of scenarios.

20.2. Denial of Service

   Denial-of-service attacks are of special concern to WebDAV servers.
   WebDAV plus HTTP enables denial-of-service attacks on every part of a
   system's resources.

   o The underlying storage can be attacked by PUTting extremely large
      files.

   o Asking for recursive operations on large collections can attack
      processing time.

   o Making multiple pipelined requests on multiple connections can
      attack network connections.

   WebDAV servers need to be aware of the possibility of a denial-of-
   service attack at all levels. The proper response to such an attack
   MAY be to simply drop the connection. Or, if the server is able to
   make a response, the server MAY use a 400-level status request such

as 400 (Bad Request) and indicate why the request was refused (a 500-
level status response would indicate that the problem is with the
server, whereas unintentional DoS attacks are something the client is
capable of remedying).

20.3. Security through Obscurity

WebDAV provides, through the PROPFIND method, a mechanism for listing
the member resources of a collection. This greatly diminishes the
effectiveness of security or privacy techniques that rely only on the
difficulty of discovering the names of network resources. Users of
WebDAV servers are encouraged to use access control techniques to
prevent unwanted access to resources, rather than depending on the
relative obscurity of their resource names.

20.4. Privacy Issues Connected to Locks

When submitting a lock request, a user agent may also submit an
'owner' XML field giving contact information for the person taking
out the lock (for those cases where a person, rather than a robot, is
taking out the lock). This contact information is stored in a DAV:
lockdiscovery property on the resource, and can be used by other

collaborators to begin negotiation over access to the resource.
However, in many cases, this contact information can be very private,
and should not be widely disseminated. Servers SHOULD limit read
access to the DAV:lockdiscovery property as appropriate.
Furthermore, user agents SHOULD provide control over whether contact
information is sent at all, and if contact information is sent,
control over exactly what information is sent.

20.5. Privacy Issues Connected to Properties

Since property values are typically used to hold information such as
the author of a document, there is the possibility that privacy
concerns could arise stemming from widespread access to a resource's
property data. To reduce the risk of inadvertent release of private
information via properties, servers are encouraged to develop access
control mechanisms that separate read access to the resource body and
read access to the resource's properties. This allows a user to
control the dissemination of their property data without overly
restricting access to the resource's contents.

   XML supports a facility known as "external entities", defined in
   Section 4.2.2 of [[REC-XML](#)], which instructs an XML processor to
   retrieve and include additional XML. An external XML entity can be
   used to append or modify the document type declaration (DTD)
   associated with an XML document. An external XML entity can also be
   used to include XML within the content of an XML document. For non-
   validating XML, such as the XML used in this specification, including
   an external XML entity is not required by XML. However, XML does
   state that an XML processor may, at its discretion, include the
   external XML entity.

   External XML entities have no inherent trustworthiness and are
   subject to all the attacks that are endemic to any HTTP GET request.
   Furthermore, it is possible for an external XML entity to modify the
   DTD, and hence affect the final form of an XML document, in the worst
   case, significantly modifying its semantics or exposing the XML
   processor to the security risks discussed in [[RFC3023](#)]. Therefore,
   implementers must be aware that external XML entities should be
   treated as untrustworthy. If a server chooses not to handle external
   XML entities, it SHOULD respond to requests containing external
   entities with the 'no-external-entities' condition code.

   There is also the scalability risk that would accompany a widely
   deployed application that made use of external XML entities. In this
   situation, it is possible that there would be significant numbers of
   requests for one external XML entity, potentially overloading any

   server that fields requests for the resource containing the external
   XML entity.

   Furthermore, there's also a risk based on the evaluation of "internal
   entities" as defined in Section 4.2.2 of [[REC-XML](#)]. A small,
   carefully crafted request using nested internal entities may require
   enormous amounts of memory and/or processing time to process. Server
   implementers should be aware of this risk and configure their XML
   parsers so that requests like these can be detected and rejected as
   early as possible.

This specification encourages the use of "A Universally Unique
Identifier (UUID) URN Namespace" ([RFC4122]) for lock tokens
(Section 6.5), in order to guarantee their uniqueness across space
and time. Version 1 UUIDs (defined in Section 4) MAY contain a
"node" field that "consists of an IEEE 802 MAC address, usually the
host address. For systems with multiple IEEE addresses, any
available one can be used". Since a WebDAV server will issue many
locks over its lifetime, the implication is that it may also be
publicly exposing its IEEE 802 address.

There are several risks associated with exposure of IEEE 802
addresses. Using the IEEE 802 address:

o It is possible to track the movement of hardware from subnet to
   subnet.

o It may be possible to identify the manufacturer of the hardware
   running a WebDAV server.

o It may be possible to determine the number of each type of
   computer running WebDAV.

This risk only applies to host-address-based UUID versions. Section
4 of [RFC4122] describes several other mechanisms for generating
UUIDs that do not involve the host address and therefore do not
suffer from this risk.

20.8. Hosting Malicious Content

HTTP has the ability to host programs that are executed on client
machines. These programs can take many forms including Web scripts,
executables, plug-in modules, and macros in documents. WebDAV does
not change any of the security concerns around these programs, yet
often WebDAV is used in contexts where a wide range of users can
publish documents on a server. The server might not have a close

trust relationship with the author that is publishing the document.
Servers that allow clients to publish arbitrary content can usefully
implement precautions to check that content published to the server
is not harmful to other clients. Servers could do this by techniques
such as restricting the types of content that is allowed to be

published and running virus and malware detection software on
published content. Servers can also mitigate the risk by having
appropriate access restriction and authentication of users that are
allowed to publish content to the server.

21. IANA Considerations

21.1. New URI Schemes

   This specification defines two URI schemes:

   1. the "opaquelocktoken" scheme defined in Appendix C, and

   2. the "DAV" URI scheme, which historically was used in [RFC2518] to
       disambiguate WebDAV property and XML element names and which
       continues to be used for that purpose in this specification and
       others extending WebDAV. Creation of identifiers in the "DAV:"
       namespace is controlled by the IETF.

   Note that defining new URI schemes for XML namespaces is now
   discouraged. "DAV:" was defined before standard best practices
   emerged.

21.2. XML Namespaces

   XML namespaces disambiguate WebDAV property names and XML elements.
   Any WebDAV user or application can define a new namespace in order to
   create custom properties or extend WebDAV XML syntax. IANA does not
   need to manage such namespaces, property names, or element names.

21.3. Message Header Fields

   The message header fields below should be added to the permanent
   registry (see [RFC3864]).

21.3.1. DAV

   Header field name: DAV

   Applicable protocol: http

   Status: standard

Author/Change controller: IETF

Specification document: this specification (Section 10.1)

21.3.2. Depth

Header field name: Depth

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document: this specification (Section 10.2)

21.3.3. Destination

Header field name: Destination

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document: this specification (Section 10.3)

21.3.4. If

Header field name: If

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document: this specification (Section 10.4)

21.3.5. Lock-Token

Header field name: Lock-Token

Applicable protocol: http

Status: standard

   Author/Change controller: IETF

   Specification document: this specification (Section 10.5)

21.3.6. Overwrite

   Header field name: Overwrite

   Applicable protocol: http

   Status: standard

   Author/Change controller: IETF

   Specification document: this specification (Section 10.6)

21.3.7. Timeout

   Header field name: Timeout

   Applicable protocol: http

   Status: standard

   Author/Change controller: IETF

   Specification document: this specification (Section 10.7)

21.4. HTTP Status Codes

   This specification defines the HTTP status codes

   o   207 Multi-Status (Section 11.1)

   o   422 Unprocessable Entity (Section 11.2),

   o   423 Locked (Section 11.3),

   o   424 Failed Dependency (Section 11.4) and

   o   507 Insufficient Storage (Section 11.5),

to be updated in the registry at
<http://www.iana.org/assignments/http-status-codes>.

Note: the HTTP status code 102 (Processing) has been removed in this
specification; its IANA registration should continue to reference RFC
2518.

22. Acknowledgements

   A specification such as this thrives on piercing critical review and
   withers from apathetic neglect. The authors gratefully acknowledge
   the contributions of the following people, whose insights were so
   valuable at every stage of our work.

   Contributors to RFC 2518

   Terry Allen, Harald Alvestrand, Jim Amsden, Becky Anderson, Alan
   Babich, Sanford Barr, Dylan Barrell, Bernard Chester, Tim Berners-
   Lee, Dan Connolly, Jim Cunningham, Ron Daniel, Jr., Jim Davis, Keith
   Dawson, Mark Day, Brian Deen, Martin Duerst, David Durand, Lee
   Farrell, Chuck Fay, Wesley Felter, Roy Fielding, Mark Fisher, Alan
   Freier, George Florentine, Jim Gettys, Phill Hallam-Baker, Dennis
   Hamilton, Steve Henning, Mead Himelstein, Alex Hopmann, Andre van der
   Hoek, Ben Laurie, Paul Leach, Ora Lassila, Karen MacArthur, Steven
   Martin, Larry Masinter, Michael Mealling, Keith Moore, Thomas Narten,
   Henrik Nielsen, Kenji Ota, Bob Parker, Glenn Peterson, Jon Radoff,
   Saveen Reddy, Henry Sanders, Christopher Seiwald, Judith Slein, Mike
   Spreitzer, Einar Stefferud, Greg Stein, Ralph Swick, Kenji Takahashi,
   Richard N. Taylor, Robert Thau, John Turner, Sankar Virdhagriswaran,
   Fabio Vitali, Gregory Woodhouse, and Lauren Wood.

   Two from this list deserve special mention. The contributions by
   Larry Masinter have been invaluable; he both helped the formation of
   the working group and patiently coached the authors along the way.
   In so many ways he has set high standards that we have toiled to
   meet. The contributions of Judith Slein were also invaluable; by
   clarifying the requirements and in patiently reviewing version after
   version, she both improved this specification and expanded our minds
   on document management.

   We would also like to thank John Turner for developing the XML DTD.

The authors of [RFC 2518](#) were Yaron Goland, Jim Whitehead, A. Faizi, Steve Carter, and D. Jensen. Although their names had to be removed due to IETF author count restrictions, they can take credit for the majority of the design of WebDAV.

Additional Acknowledgements for This Specification

Significant contributors of text for this specification are listed as contributors in the section below. We must also gratefully acknowledge Geoff Clemm, Joel Soderberg, and Dan Brotsky for hashing out specific text on the list or in meetings. Joe Hildebrand and Cullen Jennings helped close many issues. Barry Lind described an additional security consideration and Cullen Jennings provided text

for that consideration. Jason Crawford tracked issue status for this document for a period of years, followed by Elias Sinderson.

23. Contributors to This Specification

Julian Reschke
bytes GmbH
Hafenweg 16, 48155 Muenster, Germany
EMail: julian.reschke@greenbytes.de

Elias Sinderson
University of California, Santa Cruz
1156 High Street, Santa Cruz, CA 95064
EMail: elias@cse.ucsc.edu

Jim Whitehead
University of California, Santa Cruz
1156 High Street, Santa Cruz, CA 95064
EMail: ejw@soe.ucsc.edu

24. Authors of [RFC 2518](#)

Y. Y. Goland
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

EMail: yarong@microsoft.com

E. J. Whitehead, Jr.
Dept. Of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
EMail: ejw@ics.uci.edu

A. Faizi
Netscape
685 East Middlefield Road
Mountain View, CA 94043
EMail: asad@netscape.com

S. R. Carter
Novell
1555 N. Technology Way
M/S ORM F111
Orem, UT 84097-2399
EMail: srcarter@novell.com

D. Jensen
Novell
1555 N. Technology Way
M/S ORM F111
Orem, UT 84097-2399
EMail: dcjensen@novell.com

25. References

25.1. Normative References

    [REC-XML]   Bray, T., Paoli, J., Sperberg-McQueen, C., Maler,
                       E., and F. Yergeau, "Extensible Markup Language

                          (XML) 1.0 (Fourth Edition)", W3C REC-xml-20060816,
                          August 2006,
                          <http://www.w3.org/TR/2006/REC-xml-20060816/>.

   [REC-XML-INFOSET]   Cowan, J. and R. Tobin, "XML Information Set
                          (Second Edition)", W3C REC-xml-infoset-20040204,
                          February 2004, <http://www.w3.org/TR/2004/
                          REC-xml-infoset-20040204/>.

   [REC-XML-NAMES]   Bray, T., Hollander, D., Layman, A., and R. Tobin,
                          "Namespaces in XML 1.0 (Second Edition)", W3C REC-
                          xml-names-20060816, August 2006, <http://
                          www.w3.org/TR/2006/REC-xml-names-20060816/>.

   [RFC2119]   Bradner, S., "Key words for use in RFCs to
                          Indicate Requirement Levels", BCP 14, RFC 2119,
                          March 1997.

   [RFC2277]   Alvestrand, H., "IETF Policy on Character Sets and
                          Languages", BCP 18, RFC 2277, January 1998.

   [RFC2616]   Fielding, R., Gettys, J., Mogul, J., Frystyk, H.,
                          Masinter, L., Leach, P., and T. Berners-Lee,
                          "Hypertext Transfer Protocol -- HTTP/1.1",
                          RFC 2616, June 1999.

   [RFC2617]   Franks, J., Hallam-Baker, P., Hostetler, J.,
                          Lawrence, S., Leach, P., Luotonen, A., and L.
                          Stewart, "HTTP Authentication: Basic and Digest
                          Access Authentication", RFC 2617, June 1999.

   [RFC3339]   Klyne, G., Ed. and C. Newman, "Date and Time on
                          the Internet: Timestamps", RFC 3339, July 2002.

   [RFC3629]   Yergeau, F., "UTF-8, a transformation format of
                          ISO 10646", STD 63, RFC 3629, November 2003.

   [RFC3986]   Berners-Lee, T., Fielding, R., and L. Masinter,
                          "Uniform Resource Identifier (URI): Generic
                          Syntax", STD 66, RFC 3986, January 2005.

    [RFC4122]   Leach, P., Mealling, M., and R. Salz, "A
                        Universally Unique IDentifier (UUID) URN
                        Namespace", RFC 4122, July 2005.

25.2. Informative References

    [RFC2291]   Slein, J., Vitali, F., Whitehead, E., and D.
                        Durand, "Requirements for a Distributed Authoring
                        and Versioning Protocol for the World Wide Web",
                        RFC 2291, February 1998.

    [RFC2518]   Goland, Y., Whitehead, E., Faizi, A., Carter, S.,
                        and D. Jensen, "HTTP Extensions for Distributed
                        Authoring -- WEBDAV", RFC 2518, February 1999.

    [RFC2781]   Hoffman, P. and F. Yergeau, "UTF-16, an encoding
                        of ISO 10646", RFC 2781, February 2000.

    [RFC3023]   Murata, M., St. Laurent, S., and D. Kohn, "XML
                        Media Types", RFC 3023, January 2001.

    [RFC3253]   Clemm, G., Amsden, J., Ellison, T., Kaler, C., and
                        J. Whitehead, "Versioning Extensions to WebDAV
                        (Web Distributed Authoring and Versioning)",
                        RFC 3253, March 2002.

    [RFC3648]   Whitehead, J. and J. Reschke, Ed., "Web
                        Distributed Authoring and Versioning (WebDAV)
                        Ordered Collections Protocol", RFC 3648,
                        December 2003.

    [RFC3744]   Clemm, G., Reschke, J., Sedlar, E., and J.
                        Whitehead, "Web Distributed Authoring and
                        Versioning (WebDAV) Access Control Protocol",
                        RFC 3744, May 2004.

    [RFC3864]   Klyne, G., Nottingham, M., and J. Mogul,
                        "Registration Procedures for Message Header

Fields", [BCP 90](), [RFC 3864](), September 2004.

. Notes on Processing XML Elements

. Notes on Empty XML Elements

   XML supports two mechanisms for indicating that an XML element does
   not have any content. The first is to declare an XML element of the
   form <A></A>. The second is to declare an XML element of the form
   <A/>. The two XML elements are semantically identical.

. Notes on Illegal XML Processing

   XML is a flexible data format that makes it easy to submit data that
   appears legal but in fact is not. The philosophy of "Be flexible in
   what you accept and strict in what you send" still applies, but it
   must not be applied inappropriately. XML is extremely flexible in
   dealing with issues of whitespace, element ordering, inserting new
   elements, etc. This flexibility does not require extension,
   especially not in the area of the meaning of elements.

   There is no kindness in accepting illegal combinations of XML
   elements. At best, it will cause an unwanted result and at worst it
   can cause real damage.

. Example - XML Syntax Error

   The following request body for a PROPFIND method is illegal.

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

   The definition of the propfind element only allows for the allprop or
   the propname element, not both. Thus, the above is an error and must
   be responded to with a 400 (Bad Request).

   Imagine, however, that a server wanted to be "kind" and decided to
   pick the allprop element as the true element and respond to it. A
   client running over a bandwidth limited line who intended to execute
   a propname would be in for a big surprise if the server treated the
   command as an allprop.

   Additionally, if a server were lenient and decided to reply to this
   request, the results would vary randomly from server to server, with
   some servers executing the allprop directive, and others executing
   the propname directive. This reduces interoperability rather than
   increasing it.

A.4. Example - Unexpected XML Element

   The previous example was illegal because it contained two elements
   that were explicitly banned from appearing together in the propfind
   element. However, XML is an extensible language, so one can imagine
   new elements being defined for use with propfind. Below is the
   request body of a PROPFIND and, like the previous example, must be
   rejected with a 400 (Bad Request) by a server that does not
   understand the expired-props element.

       <?xml version="1.0" encoding="utf-8" ?>
       <D:propfind xmlns:D="DAV:"
       xmlns:E="http://www.example.com/standards/props/">
        <E:expired-props/>
       </D:propfind>

   To understand why a 400 (Bad Request) is returned, let us look at the
   request body as the server unfamiliar with expired-props sees it.

       <?xml version="1.0" encoding="utf-8" ?>
       <D:propfind xmlns:D="DAV:"
                   xmlns:E="http://www.example.com/standards/props/">
       </D:propfind>

   As the server does not understand the 'expired-props' element,
   according to the WebDAV-specific XML processing rules specified in
   Section 17, it must process the request as if the element were not
   there. Thus, the server sees an empty propfind, which by the
   definition of the propfind element is illegal.

   Please note that had the extension been additive, it would not
   necessarily have resulted in a 400 (Bad Request). For example,
   imagine the following request body for a PROPFIND:

       <?xml version="1.0" encoding="utf-8" ?>
       <D:propfind xmlns:D="DAV:"
                   xmlns:E="http://www.example.com/standards/props/">
        <D:propname/>
        <E:leave-out>*boss*</E:leave-out>
       </D:propfind>

   The previous example contains the fictitious element leave-out. Its

purpose is to prevent the return of any property whose name matches the submitted pattern. If the previous example were submitted to a server unfamiliar with 'leave-out', the only result would be that the 'leave-out' element would be ignored and a propname would be executed.

_____

[Appendix B](#).   Notes on HTTP Client Compatibility

   WebDAV was designed to be, and has been found to be, backward-
   compatible with HTTP 1.1. The PUT and DELETE methods are defined in
   HTTP and thus may be used by HTTP clients as well as WebDAV-aware
   clients, but the responses to PUT and DELETE have been extended in
   this specification in ways that only a WebDAV client would be
   entirely prepared for. Some theoretical concerns were raised about
   whether those responses would cause interoperability problems with
   HTTP-only clients, and this section addresses those concerns.

   Since any HTTP client ought to handle unrecognized 400-level and 500-
   level status codes as errors, the following new status codes should
   not present any issues: 422, 423, and 507 (424 is also a new status
   code but it appears only in the body of a Multistatus response.) So,
   for example, if an HTTP client attempted to PUT or DELETE a locked
   resource, the 423 Locked response ought to result in a generic error
   presented to the user.

   The 207 Multistatus response is interesting because an HTTP client
   issuing a DELETE request to a collection might interpret a 207
   response as a success, even though it does not realize the resource
   is a collection and cannot understand that the DELETE operation might
   have been a complete or partial failure. That interpretation isn't
   entirely justified, because a 200-level response indicates that the
   server "received, understood, and accepted" the request, not that the
   request resulted in complete success.

   One option is that a server could treat a DELETE of a collection as
   an atomic operation, and use either 204 No Content in case of
   success, or some appropriate error response (400 or 500 level) for an
   error. This approach would indeed maximize backward compatibility.
   However, since interoperability tests and working group discussions
   have not turned up any instances of HTTP clients issuing a DELETE
   request against a WebDAV collection, this concern is more theoretical
   than practical. Thus, servers are likely to be completely successful

at interoperating with HTTP clients even if they treat any collection
DELETE request as a WebDAV request and send a 207 Multi-Status
response.

In general, server implementations are encouraged to use the detailed
responses and other mechanisms defined in this document rather than
make changes for theoretical interoperability concerns.

Appendix C. The 'opaquelocktoken' Scheme and URIs

The 'opaquelocktoken' URI scheme was defined in [RFC2518] (and
registered by IANA) in order to create syntactically correct and
easy-to-generate URIs out of UUIDs, intended to be used as lock
tokens and to be unique across all resources for all time.

An opaquelocktoken URI is constructed by concatenating the
'opaquelocktoken' scheme with a UUID, along with an optional
extension. Servers can create new UUIDs for each new lock token. If
a server wishes to reuse UUIDs, the server MUST add an extension, and
the algorithm generating the extension MUST guarantee that the same
extension will never be used twice with the associated UUID.

```
OpaqueLockToken-URI = "opaquelocktoken:" UUID [Extension]
  ; UUID is defined in Section 3 of [RFC4122].  Note that LWS
  ; is not allowed between elements of
  ; this production.

Extension = path
  ; path is defined in Section 3.3 of [RFC3986]
```

Appendix D. Lock-null Resources

The original WebDAV model for locking unmapped URLs created "lock-
null resources". This model was over-complicated and some
interoperability and implementation problems were discovered. The
new WebDAV model for locking unmapped URLs (see Section 7.3) creates

"locked empty resources". Lock-null resources are deprecated. This
section discusses the original model briefly because clients MUST be
able to handle either model.

In the original "lock-null resource" model, which is no longer
recommended for implementation:

o A lock-null resource sometimes appeared as "Not Found". The
   server responds with a 404 or 405 to any method except for PUT,
   MKCOL, OPTIONS, PROPFIND, LOCK, UNLOCK.

o A lock-null resource does however show up as a member of its
   parent collection.

o The server removes the lock-null resource entirely (its URI
   becomes unmapped) if its lock goes away before it is converted to
   a regular resource. Recall that locks go away not only when they
   expire or are unlocked, but are also removed if a resource is
   renamed or moved, or if any parent collection is renamed or moved.

o The server converts the lock-null resource into a regular resource
   if a PUT request to the URL is successful.

o The server converts the lock-null resource into a collection if a
   MKCOL request to the URL is successful (though interoperability
   experience showed that not all servers followed this requirement).

o Property values were defined for DAV:lockdiscovery and DAV:
   supportedlock properties but not necessarily for other properties
   like DAV:getcontenttype.

Clients can easily interoperate both with servers that support the
old model "lock-null resources" and the recommended model of "locked
empty resources" by only attempting PUT after a LOCK to an unmapped
URL, not MKCOL or GET.

[D.1](#). Guidance for Clients Using LOCK to Create Resources

A WebDAV client implemented to this specification might find servers
that create lock-null resources (implemented before this
specification using [[RFC2518](#)]) as well as servers that create locked
empty resources. The response to the LOCK request will not indicate

what kind of resource was created. There are a few techniques that
help the client deal with either type.

    If the client wishes to avoid accidentally creating either lock-
    null or empty locked resources, an "If-Match: *" header can be
    included with LOCK requests to prevent the server from creating a
    new resource.

    If a LOCK request creates a resource and the client subsequently
    wants to overwrite that resource using a COPY or MOVE request, the
    client should include an "Overwrite: T" header.

    If a LOCK request creates a resource and the client then decides
    to get rid of that resource, a DELETE request is supposed to fail
    on a lock-null resource and UNLOCK should be used instead. But
    with a locked empty resource, UNLOCK doesn't make the resource
    disappear. Therefore, the client might have to try both requests
    and ignore an error in one of the two requests.

[Appendix E](). Guidance for Clients Desiring to Authenticate

   Many WebDAV clients that have already been implemented have account
   settings (similar to the way email clients store IMAP account
   settings). Thus, the WebDAV client would be able to authenticate
   with its first couple requests to the server, provided it had a way
   to get the authentication challenge from the server with realm name,

   nonce, and other challenge information. Note that the results of
   some requests might vary according to whether or not the client is
   authenticated -- a PROPFIND might return more visible resources if
   the client is authenticated, yet not fail if the client is anonymous.

   There are a number of ways the client might be able to trigger the
   server to provide an authentication challenge. This appendix
   describes a couple approaches that seem particularly likely to work.

   The first approach is to perform a request that ought to require
   authentication. However, it's possible that a server might handle
   any request even without authentication, so to be entirely safe, the
   client could add a conditional header to ensure that even if the
   request passes permissions checks, it's not actually handled by the
   server. An example of following this approach would be to use a PUT

request with an "If-Match" header with a made-up ETag value. This
approach might fail to result in an authentication challenge if the
server does not test authorization before testing conditionals as is
required (see [Section 8.5](#)), or if the server does not need to test
authorization.

Example - forcing auth challenge with write request

>>Request

      PUT /forceauth.txt HTTP/1.1
      Host: www.example.com
      If-Match: "xxx"
      Content-Type: text/plain
      Content-Length: 0

The second approach is to use an Authorization header (defined in
[[RFC2617](#)]), which is likely to be rejected by the server but which
will then prompt a proper authentication challenge. For example, the
client could start with a PROPFIND request containing an
Authorization header containing a made-up Basic userid:password
string or with actual plausible credentials. This approach relies on
the server responding with a "401 Unauthorized" along with a
challenge if it receives an Authorization header with an unrecognized
username, invalid password, or if it doesn't even handle Basic
authentication. This seems likely to work because of the
requirements of [RFC 2617](#):

   "If the origin server does not wish to accept the credentials sent
   with a request, it SHOULD return a 401 (Unauthorized) response. The
   response MUST include a WWW-Authenticate header field containing at
   least one (possibly new) challenge applicable to the requested
   resource."

   There's a slight problem with implementing that recommendation in
   some cases, because some servers do not even have challenge

information for certain resources. Thus, when there's no way to
authenticate to a resource or the resource is entirely publicly
available over all accepted methods, the server MAY ignore the
Authorization header, and the client will presumably try again later.

Example - forcing auth challenge with Authorization header

>>Request

```
PROPFIND /docs/ HTTP/1.1
Host: www.example.com
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Content-type: application/xml; charset="utf-8"
Content-Length: xxxx

[body omitted]
```

Appendix F. Summary of Changes from RFC 2518

This section lists major changes between this document and RFC 2518,
starting with those that are likely to result in implementation
changes. Servers will advertise support for all changes in this
specification by returning the compliance class "3" in the DAV
response header (see Sections 10.1 and 18.3).

F.1. Changes for Both Client and Server Implementations

Collections and Namespace Operations

o   The semantics of PROPFIND 'allprop' (Section 9.1) have been
    relaxed so that servers return results including, at a minimum,
    the live properties defined in this specification, but not
    necessarily return other live properties. The 'allprop' directive
    therefore means something more like "return all properties that
    are supposed to be returned when 'allprop' is requested" -- a set
    of properties that may include custom properties and properties
    defined in other specifications if those other specifications so
    require. Related to this, 'allprop' requests can now be extended
    with the 'include' syntax to include specific named properties,

thereby avoiding additional requests due to changed 'allprop'

semantics.

o Servers are now allowed to reject PROPFIND requests with Depth:
   Infinity. Clients that used this will need to be able to do a
   series of Depth:1 requests instead.

o Multi-Status response bodies now can transport the value of HTTP's
   Location response header in the new 'location' element. Clients
   may use this to avoid additional roundtrips to the server when
   there is a 'response' element with a 3xx status (see
   Section 14.24).

o The definition of COPY has been relaxed so that it doesn't require
   servers to first delete the target resources anymore (this was a
   known incompatibility with [RFC3253]). See Section 9.8.

Headers and Marshalling

o The Destination and If request headers now allow absolute paths in
   addition to full URIs (see Section 8.3). This may be useful for
   clients operating through a reverse proxy that does rewrite the
   Host request header, but not WebDAV-specific headers.

o This specification adopts the error marshalling extensions and the
   "precondition/postcondition" terminology defined in [RFC3253] (see
   Section 16). Related to that, it adds the "error" XML element
   inside multistatus response bodies (see Section 14.5, however note
   that it uses a format different from the one recommended in RFC
   3253).

o Senders and recipients are now required to support the UTF-16
   character encoding in XML message bodies (see Section 19).

o Clients are now required to send the Depth header on PROPFIND
   requests, although servers are still encouraged to support clients
   that don't.

Locking

o RFC 2518's concept of "lock-null resources" (LNRs) has been
   replaced by a simplified approach, the "locked empty resources"
   (see Section 7.3). There are some aspects of lock-null resources
   clients cannot rely on anymore, namely, the ability to use them to
   create a locked collection or the fact that they disappear upon
   UNLOCK when no PUT or MKCOL request was issued. Note that servers
   are still allowed to implement LNRs as per RFC 2518.

o   There is no implicit refresh of locks anymore. Locks are only
    refreshed upon explicit request (see Section 9.10.2).

o   Clarified that the DAV:owner value supplied in the LOCK request
    must be preserved by the server just like a dead property
    (Section 14.17). Also added the DAV:lockroot element
    (Section 14.12), which allows clients to discover the root of
    lock.

F.2. Changes for Server Implementations

    Collections and Namespace Operations

o   Due to interoperability problems, allowable formats for contents
    of 'href' elements in multistatus responses have been limited (see
    Section 8.3).

o   Due to lack of implementation, support for the 'propertybehavior'
    request body for COPY and MOVE has been removed. Instead,
    requirements for property preservation have been clarified (see
    Sections 9.8 and 9.9).

    Properties

o   Strengthened server requirements for storage of property values,
    in particular persistence of language information (xml:lang),
    whitespace, and XML namespace information (see Section 4.3).

o   Clarified requirements on which properties should be writable by
    the client; in particular, setting "DAV:displayname" should be
    supported by servers (see Section 15).

o   Only 'rfc1123-date' productions are legal as values for DAV:
    getlastmodified (see Section 15.7).

    Headers and Marshalling

o   Servers are now required to do authorization checks before
    processing conditional headers (see Section 8.5).

    Locking

o   Strengthened requirement to check identity of lock creator when
    accessing locked resources (see Section 6.4). Clients should be
    aware that lock tokens returned to other principals can only be
    used to break a lock, if at all.

   o Section 8.10.4 of [RFC2518] incorrectly required servers to return
      a 409 status where a 207 status was really appropriate. This has
      been corrected (Section 9.10).

F.3. Other Changes

   The definition of collection state has been fixed so it doesn't vary
   anymore depending on the Request-URI (see Section 5.2).

   The DAV:source property introduced in Section 4.6 of [RFC2518] was
   removed due to lack of implementation experience.

   The DAV header now allows non-IETF extensions through URIs in
   addition to compliance class tokens. It also can now be used in
   requests, although this specification does not define any associated
   semantics for the compliance classes defined in here (see
   Section 10.1).

   In RFC 2518, the definition of the Depth header (Section 9.2)
   required that, by default, request headers would be applied to each
   resource in scope. Based on implementation experience, the default
   has now been reversed (see Section 10.2).

   The definitions of HTTP status code 102 ([RFC2518], Section 10.1) and
   the Status-URI response header (Section 9.7) have been removed due to
   lack of implementation.

   The TimeType format used in the Timeout request header and the
   "timeout" XML element used to be extensible. Now, only the two
   formats defined by this specification are allowed (see Section 10.7).

Author's Address

   Lisa Dusseault (editor)
   CommerceNet
   2064 Edgewood Dr.
   Palo Alto, CA 94303
   US

EMail: ldusseault@commerce.net