WebDAV
Internet-Draft
Obsoletes: 2518 (if approved)
Expires: July 3, 2006

L. Dusseault
OSAF

December 30, 2005

HTTP Extensions for Distributed Authoring - WebDAV
draft-ietf-webdav-rfc2518bis-10

Status of this Memo

Copyright Notice

Abstract

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).

RFC2518 was published in February 1999, and this specification makes 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. Also, the ability to link pages of any media type to related pages.

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 which change the URL.

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 standard 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 8) and the new HTTP headers (Section 9) used with WebDAV methods.

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 new status codes developed for WebDAV methods (Section 10) and describes existing HTTP status codes (Section 11) as used in WebDAV. Since some WebDAV methods may operate over many resources, the Multi-Status response (Section 12)

has been introduced to return status information for multiple resources. Finally, this version of WebDAV introduces precondition and postcondition (Section 15) XML elements in error response bodies.

WebDAV uses [XML] for property names and some values, and also uses XML to marshal complicated request and response. This specification contains DTD and text definitions of all all properties (Section 14) and all other XML elements (Section 13) used in marshalling. WebDAV includes a few special rules on how to process XML (Section 16) appearing in WebDAV so that it truly is extensible.

Finishing off the specification are sections on what it means for a resource to be compliant with this specification (Section 17), on internationalization support (Section 18), and on security (Section 19).

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 white-space. Since this augmented BNF uses the basic production rules provided in section 2.2 of [RFC2616], these rules apply to this document as well.

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.

Collection - A resource that contains a set of URLs, which identify and locate member resources and which meet the collections requirements (Section 5).

Member URL - A URL which is a member of the set of URLs contained by a collection.

Internal Member URL - A Member URL that is immediately relative to the URL of the collection (the definition of immediately relative is given later (Section 5.2)).

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 "principal" is a distinct human or computational actor that initiates access to network resources.

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 read-only, 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 which 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. XML Usage

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] 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 MIME 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.

The XML namespace extension [W3C.REC-xml-names-19990114] is also 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.

Note that "DAV:" uses a scheme name defined solely for the purpose of creating this XML namespace. Defining new URI schemes for namespaces is discouraged. "DAV:" was defined before standard best practices emerged, and this namespace is still used only because of significant existing deployments.

4.4. 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 new elements. Older 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 a 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), not multiple values in different languages or a single value in multiple languages.

A property is always represented in XML 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 a 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. In the latter case, servers MUST preserve certain aspects of the content (described using the terminology from [W3C.REC-xml-infoset-20040204]).

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 value:

[namespace name]

[local name]

[attributes]

[children] of type element or character

On Attribute Information Items in the value:

[namespace name]

[local name]

[normalized value]

On Character Information Items in the value:

[character code]

Since prefixes are used in some XML query/handling tools, servers SHOULD preserve, for any Information Item in the value:

[prefix]

In dead properties (considered as content, like document bodies) servers are encouraged to (MAY) preserve, for any Comment Information Item in the value:

[content]

XML Infoset attributes not listed above MAY be preserved by the server, but clients MUST NOT rely on them being preserved.

The XML attribute xml:space MUST NOT be used to change white space handling. White space in property values is significant.

4.4.1. Example - Property with Mixed Content

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

```
<D:prop xml:lang="en">
  <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><author xmlns:x='http://example.com/ns' xml:lang="en"
        xmlns='http://example.com/ns'
        xmlns:ns1='http://www.w3.org/1999/xhtml'>
    <x:name>Jane Doe</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:

- The [prefix] for the property name itself was not preserved, being non-significant
- attribute values have been rewritten with double quotes instead of single quotes (quoting style is not significant), and attribute order has not been preserved,
- 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),
- whitespace between tags has been preserved everywhere (whitespace between attributes not so),
- CDATA encapsulation was replaced with character escaping (the reverse would also be legal),
- 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.

4.5. 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 which 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.

4.6. 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 new type of Web resource, the collection, and discusses its interactions with the HTTP URL namespace. 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. 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 require 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.

Although 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

A collection is a resource whose state consists of at least a list of internal member URLs and a set of properties, but which may have additional state such as entity bodies returned by GET. An internal member URL MUST be immediately relative to a base URL of the collection. That is, the internal member URL is equal to a containing collection's URL plus an additional segment for non-collection resources, or additional segment plus trailing slash "/" for collection resources, where segment is defined in section 3.3 of [RFC3986].

Any given internal member URL MUST only belong to the collection once, i.e., it is illegal to have multiple instances of the same URL

in a collection. Properties defined on collections behave exactly as do properties on non-collection resources.

For all WebDAV compliant resources A and B, identified by URLs U and V, for which U is immediately relative to V, B MUST be a collection that has U as an internal member URL. So, if the resource with URL http://example.com/bar/blah is WebDAV compliant and if the resource with URL http://example.com/bar/ is WebDAV compliant then the resource with URL http://example.com/bar/ must be a collection and must contain URL http://example.com/bar/blah as an internal member.

Collection resources MAY list the URLs of non-WebDAV compliant children in the HTTP URL namespace hierarchy as internal members but are not required to do so. For example, if the resource with URL http://example.com/bar/blah is not WebDAV compliant and the URL http://example.com/bar/ identifies a collection then URL http://example.com/bar/blah may or may not be an internal member of the collection with URL http://example.com/bar/.

If a WebDAV compliant resource has no WebDAV compliant children 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 a 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.

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. Exclusive Vs. Shared Locks

The most basic form of lock is an exclusive lock. Only one exclusive lock may exist on any resource, whether it is directly or indirectly locked (Section 7.7). Exclusive locks avoid having to merge results, 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 with appropriate access can use the lock.

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 owner leaves without unlocking a resource. While both timeouts 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.

6.2. 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.

6.3. Lock Tokens

A lock token is a type of state token, represented as a URI, which 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 9.5, and also in the body of the response.

Submitting a lock token does not confer full privilege to use the lock token or 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.

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 tokens created by another client instance.

This specification encourages servers to create 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"

6.4. 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.

6.5. 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.

6.6. Locks and Multiple Bindings

A resource may be made available through more than one URI. A lock MUST cover the resource as well as the URI to which the LOCK request was addressed. The lock MAY cover other URIs mapped to the same resource as well.

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 will prevent parallel changes to a resource by any principal other than the write lock holder. In general terms, changes affected by write locks include changes to:

- the content of the resource
- any dead property of the resource
- any live property defined to be lockable (all properties defined in this specification are lockable)
- the direct membership of the resource, if it is a collection
- the URL/location of a resource

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

7.1. Lock Owner

The creator of the lock is the lock owner. The server MUST restrict the usage of the lock token to the lock owner (both for shared and exclusive locks -- for multi-user shared lock cases, each authenticated principal MUST obtain its own shared lock).

The server MAY allow privileged users other than the lock owner to destroy a lock (for example, the resource owner or an administrator) as a special case of lock usage.

If an anonymous user requests a lock, the server MAY refuse the request.

7.2. Methods Restricted by Write Locks

A server MUST reject any write request that alters a write-locked resource unless a valid lock token is provided. The write operations defined in HTTP and WebDAV are PUT, POST, PROPPATCH, LOCK, UNLOCK, MOVE, COPY (for the destination resource), DELETE, and MKCOL. All other HTTP/WebDAV methods, GET in particular, function independently of the lock. A shared write lock prevents the same operations (except additional requests for shared write locks), however it also allows access by any principal that has a shared write lock on the

same resource.

Note, however, that as new methods are created it will be necessary to specify how they interact with a write lock.

7.3. Write Locks and Lock Tokens

A successful request for an exclusive or shared write lock MUST result in the generation of a unique lock token associated with the requesting principal. Thus if five principals have a shared write lock on the same resource there will be five lock tokens, one for each principal.

7.4. 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 to respect locks are guaranteed not to change while write locked.

7.5. 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.

7.6. Write Locks and Unmapped URLs

WebDAV provides the ability to lock 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 HTTP 1.1). 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 may 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 an locked resource with empty content. Subsequently, a successful PUT request (with the correct lock token) provides the content for the resource, and the server MUST also use the content-type and content-language information from this request.

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 creates "locked empty resources". Servers MUST implement either lock-null resources or locked empty resources, but servers SHOULD implement locked empty resources. This section discusses the original model briefly and the

new model more completely, because clients MUST be able to handle either model.

In the original "lock-null resource" model, which is no longer recommended for implementation:

- 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.
- A lock-null resource does however show up as a member of its parent collection.
- 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 unlcoked, but are also removed if a resource is renamed or moved, or if any parent collection is renamed or moved.
- The server converts the lock-null resource into a regular resource if a PUT request to the URL is successful.
- 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).
- Property values were defined for DAV:lockdiscovery and DAV: supportedlock properties but not necessarily for other properties like DAV:getcontenttype.

In the "locked empty resource" model, which is now the recommended implementation, a resource created with a LOCK is empty but otherwise behaves in every way as a normal resource. A locked empty resource:

- Can be read, deleted, moved, copied, and in all ways behave as a regular resource, not a lock-null resource.
- Appears as a member of its parent collection.
- 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)
- SHOULD default to having no content type.
- MAY NOT have values for properties like DAV:getcontentlanguage which haven't been specified yet by the client.

- Can be updated (have content added) with a PUT request. The server MUST be able to set the content type as specified in the PUT request.

- 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).

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

- 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. When the client uses PUT to overwrite a locked empty resource the client MUST supply a Content-Type if any is known. If the client supplies a Content-Type value the server MUST set that value (this requirement actually applies to any resource that is overwritten but is particularly necessary for locked empty resources which are initially created with no Content-Type).

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.

7.7. Write Locks and Collections

A write lock on a collection, whether created by a "Depth: 0" or "Depth: infinity" lock request, prevents the addition or removal of member URLs of the collection by non-lock owners.

A zero-depth lock on a collection affects changes to the direct membership of that collection. When a principal issues a write request to create a new resource in a write locked collection, or isses a DELETE, MOVE or other request that would remove an existing internal member URL of a write locked collection or change the binding name, this request MUST fail if the principal does not provide the correct lock token for the locked collection.

This means that if a collection is locked (depth 0 or infinity), its lock-token is required in all these cases:

- DELETE a collection's direct internal member
- MOVE a member out of the collection
- MOVE a member into the collection
- MOVE to rename a member within a collection
- COPY a member into a collection
- PUT or MKCOL request which would create a new 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 descendents of the locked collection. With a depth-infinity lock, the root of the lock is directly locked, and all its descendants are indirectly locked.

- Any new resource added as a descendent of a depth-infinity locked collection becomes indirectly locked.
- Any indirectly locked resource moved out of the locked collection into an unlocked collection is thereafter unlocked.
- Any indirectly locked resource moved out of a locked source collection into a depth-infinity locked target collection remains indirectly locked but is now within the scope of 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 which conflicts with the write lock, the request MUST fail with a 423 (Locked) status code, and the response SHOULD contain the 'lock-token-present' precondition.

If a lock owner 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 added to the lock. This is the only mechanism that allows a resource to be added to a write lock. Thus, 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.

7.8. Write Locks and the If Request Header

If a user agent is not required to have knowledge about a lock when requesting an operation on a locked resource, the following scenario might occur. 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.

7.8.1. Example - Write Lock

>>Request

```
COPY /~fielding/index.html HTTP/1.1
Host: www.ics.uci.edu
Destination: http://www.ics.uci.edu/users/f/fielding/index.html
If: <http://www.ics.uci.edu/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, 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.

7.9. 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 "Depth: infinity", 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 is locked with "Depth: infinity", then the resource will be added to that collection's lock. Additionally, if a resource locked with "Depth: infinity" 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 a added to the lock. In both these examples, as specified in Section 7.8, an If header must be submitted containing a lock token for both the source and destination.

7.10. 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 method with an If header but without a body. This form of LOCK MUST only be used to "refresh" a lock. Meaning, at minimum, that any timers associated with the lock MUST be re-set.

A server may return a Timeout header with a lock refresh that is different than the Timeout header returned when the lock was originally requested. Additionally clients may submit Timeout headers of arbitrary value with their lock refresh requests. Servers, as always, may ignore Timeout headers submitted by the client. Note that timeout is measured in seconds remaining until expiration.

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

8. HTTP Methods for Distributed Authoring

8.1. General Request and Response Handling

8.1.1. Use of XML

Some of the following new HTTP methods use XML as a request and response format. All DAV compliant clients and resources MUST use XML parsers that are compliant with [XML] and XML Namespaces [W3C.REC-xml-names-19990114]. 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 19). 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.

8.1.2. 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 they intended.

8.1.3. 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.

8.1.4. ETag

HTTP 1.1 recommends the use of the ETag header in responses to GET and PUT requests. 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 whether it has changed. Timestamps do not solve this problem nearly as well as ETags.

WebDAV servers SHOULD support strong ETags for all resources that may be PUT. If ETags are supported for a resource, the server MUST return the ETag header in all PUT and GET responses to that resource.

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.

8.1.5. Including error response bodies

HTTP and WebDAV did not use the bodies of most error responses for machine-parsable information until DeltaV 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 15

8.2. PROPFIND

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 13.20) along with all XML elements defined for use with that element.

A client may submit a Depth header with a value of "0", "1", or

"infinity" with a PROPFIND on a collection resource. Servers MUST support the "0", "1" and "infinity" behaviors on WebDAV-compliant resources. By default, the PROPFIND method without a Depth header MUST act 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:

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

- Request all dead property values, by using 'dead-props' element. This can be combined with retrieving specific live properties named as above. Servers advertising support for this specification MUST support this feature.

- Request property values for those properties defined in this specification plus dead properties, by using 'allprop' element

- 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. A WebDAV server MAY omit certain live properties from other specifications when responding to an 'allprop' request from an older client, and MAY return only custom (dead) properties and those defined in this specification.

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 which does not exist is an error and MUST be noted, if the response uses a 'multistatus' XML element, with a 'response' XML element which contains a 404 (Not Found) status value.

Consequently, the 'multistatus' XML element for a collection resource with member URLs MUST include a 'response' XML element for each member URL of the collection, to whatever depth was requested. Each 'response' XML element MUST contain an 'href' XML 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 with internal member URLs are returned as a flat list whose order of entries is not significant.

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.

The results of this method SHOULD NOT be cached.

8.2.1. 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 15) that might be particularly useful with PROPFIND.

403 Forbidden - A server MAY reject all 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.

8.2.2. Status codes for use with 207 (Multi-Status)

The following status codes are defined for use within the PROPFIND Multi-Status response:

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.

8.2.3. 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://www.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://www.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.

8.2.4. Example - Retrieving Named and Dead Properties

>>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:prop>
    <D:creationdate/>
    <D:getlastmodified/>
  </D:prop>
  <D:dead-props/>
</D:propfind>
```

In this example, PROPFIND is executed on a collection resource http://www.example.com/mycol/. The client requests the values of two specific live properties plus all dead properties (names and values). The response is not shown.

8.2.5. 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://www.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://www.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 descendents 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://www.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://www.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 which do not explicitly state the namespace to which they belong are members of the "DAV:" namespace.

8.2.6. Example - Using '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://www.foo.bar/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://www.foo.bar/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
              >Monday, 12-Jan-98 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.foo.bar/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.foo.bar/container/ has six properties defined on it: 'bigbox' and 'author in the "http://www.foo.bar/boxschema/" namespace, DAV:creationdate, DAV:displayname, DAV:resourcetype, and DAV:supportedlock.

The last four properties are WebDAV-specific, defined in Section 14. 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.foo.bar/container/front.html has nine properties defined on it:

'bigbox' in the "http://www.foo.bar/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).

8.3. PROPPATCH

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. Instruction processing MUST occur 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 Section 13.23 and Section 13.26.

8.3.1. Status Codes for use in 207 (Multi-Status)

The following are examples of response codes one would expect to be used in a 207 (Multi-Status) response for this method. Note, however, that unless explicitly prohibited any 2/3/4/5xx series response code may be used in a 207 (Multi-Status) response.

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 read-only property, such as DAV:getetag. If returning this error, the server SHOULD use the precondition code 'writable-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.

8.3.2. 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://www.w3.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://www.w3.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 can not 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://www.w3.com/standards/z39.50/" namespace, and to remove the property "Copyright-Owner" in the "http://www.w3.com/standards/z39.50/" 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.

8.4. MKCOL Method

The MKCOL method is used to create a new collection. All WebDAV compliant resources MUST support the 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 a 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.

8.4.1. MKCOL Status Codes

Responses from a MKCOL request MUST NOT be cached as MKCOL has non-idempotent semantics. 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 (since this specification does not define any body for MKCOL requests).

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

8.4.2. 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
```

8.5. 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.

8.6. 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.

8.7. DELETE

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).

8.7.1. 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 an internal 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.

8.7.2. 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: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 can not 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.

8.8. PUT

8.8.1. 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).

8.8.2. 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.

8.9. COPY

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.

8.9.1. 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.

8.9.2. COPY for Properties

After a successful COPY invocation, all dead properties on the source resource MUST be duplicated on the destination resource, along with all properties as appropriate. 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.

A COPY operation creates a new resource, much like a PUT operation does. Live properties which are related to resource creation (such as DAV:creationdate) should have their values set accordingly.

8.9.3. 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.

A COPY of depth infinity 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 a depth infinity 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 descendents 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.

8.9.4. 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.

8.9.5. 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 pre-existing 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. E.g. 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-present' 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.

8.9.6. Example - COPY with Overwrite

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

```
>>Request

  COPY /~fielding/index.html HTTP/1.1
  Host: www.ics.uci.edu
  Destination: http://www.ics.uci.edu/users/f/fielding/index.html

>>Response

  HTTP/1.1 204 No Content
```

8.9.7. 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.ics.uci.edu
  Destination: http://www.ics.uci.edu/users/f/fielding/index.html
  Overwrite: F

>>Response

  HTTP/1.1 412 Precondition Failed
```

8.9.8. 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: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.

8.10. MOVE

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 atomically. The consistency maintenance step allows the server to perform updates caused by the move, such as updating all URLs other than the Request-URI which identify the source resource, to point to the new destination resource. Consequently, 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. However, 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.

8.10.1. 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 which 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.

8.10.2. 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 section 5.1 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 descendents 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.

8.10.3. 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.

8.10.4. 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. E.g. 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 state of 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-present' 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.

8.10.5. Example - MOVE of a Non-Collection

This example shows resource http://www.ics.uci.edu/~fielding/index.html being moved to the location http://www.ics.uci.edu/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.ics.uci.edu
Destination: http://www.ics.uci.edu/users/f/fielding/index.html
```

>>Response

```
HTTP/1.1 201 Created
Location: http://www.ics.uci.edu/users/f/fielding/index.html
```

8.10.6. 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: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.

8.11. 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 which supports the LOCK method MUST, at minimum, support the XML request and response formats defined herein.

A LOCK method invocation to an unlocked resource creates a lock on the resource identified by the Request-URI, which becomes the root of the lock. Lock method requests to create a new lock MUST have a XML request body which contains an owner XML element and other information for this lock request. The server MUST preserve the information provided by the client in the 'owner' field when the lock information is requested. The LOCK request MAY have a Timeout header.

Clients MUST assume that locks may 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.

When a new lock is created, the LOCK response:

- 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.

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

8.11.1. Refreshing Locks

A lock is refreshed by sending a LOCK request without a request body to the URL of a resource within the scope of the lock. This request MUST specify which lock to refresh by using the 'Lock-Token' header with a single lock token (only one lock may be refreshed at a time). It 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.

Note that in RFC2518, clients were indicated through the example in the text to use the If header to specify what lock to refresh (rather than the Lock-Token header). Servers are encouraged to continue to support this as well as the Lock-Token header.

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

8.11.2. 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 internal members, all the way down the hierarchy, are to be locked. A successful result MUST return a single lock token which represents all the resources that have been locked. If an UNLOCK is successfully executed on this token, all associated resources are unlocked. Hence, partial success is not an option. 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 which 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 MUST 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.

8.11.3. Locking Unmapped URLs

A successful LOCK method MUST result in the creation of an empty resource which is locked (and which 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 and no Content-Type.

8.11.4. 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 the lock must not be granted.

8.11.5. 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) - The resource is locked already.

400 (Bad Request), with 'lock-token-matches-request-uri' precondition code - The LOCK request was made with a Lock-Token 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.

8.11.6. 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://www.ics.uci.edu/~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://www.ics.uci.edu/~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://www.ics.uci.edu/~ejw/contact.html contains contact information for the owner 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.

8.11.7. Example - Refreshing a Write Lock

```
>>Request

  LOCK /workspace/webdav/proposal.doc HTTP/1.1
  Host: example.com
  Timeout: Infinite, Second-4100000000
  Lock-Token: <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://www.ics.uci.edu/~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.

8.11.8. 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://www.ics.uci.edu/~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.

8.12. 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 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 remove the lock from the resource identified by the Request-URI and from all other resources included in the lock.

If all resources which have been locked under the submitted lock token can not 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 which supports the LOCK method MUST support the UNLOCK method.

8.12.1. 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 (see 'lock-token-present' precondition), or request was made to a Request-URI that was

not within the scope of the lock (see 'lock-token-matches-request-uri' precondition).

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

409 (Conflict) - The resource was not locked and thus could not be unlocked.

8.12.2. 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. The 204 (No Content) status code is used instead of 200 (OK) because there is no response entity body.

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

9. 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.

9.1. DAV Header

```
DAV             = "DAV" ":" #( compliance-class )
compliance-class = ( "1" | "2" | "bis" | extend )
extend          = Coded-URL | token
Coded-URL       = "<" absolute-URI ">"
                    ; No LWS allowed in Coded-URL
                    ; absolute-URI is defined in RFC3986
```

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.

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 "bis" compliance. In general, support for one compliance class does not entail support for any other. Please refer to section 16 for more details on compliance classes defined in this specification.

This header must also appear on responses to OPTIONS requests to the special '*' Request-URI as defined in HTTP/1.1. In this case it means that the repository supports the named features in at least some internal URL namespaces.

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.

9.2. Depth Header

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

The Depth request header is used with methods executed on resources which 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 immediate children, ("Depth: 1"), or the resource and all its progeny ("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 which 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 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.

Any headers on a method that has a defined interaction with the Depth header MUST be applied to all resources in the scope of the method except where alternative behavior is explicitly defined. For example, an If-Match header will have its value applied against every resource in the method's scope and will cause the method to fail if the header fails to match.

If a resource, source or destination, within the scope of the method with a 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 children. If a resource does not have internal children then the Depth header MUST be ignored.

Please note, however, that it is always an error to submit a value for the Depth header that is not allowed by the method's definition. Thus submitting a "Depth: 1" on a COPY, even if the resource does not have internal members, will result in a 400 (Bad Request). The method should fail not because the resource doesn't have internal members, but because of the illegal value in the header.

9.3. Destination Header

```
Destination = "Destination" ":" ( absolute-URI )
```

The Destination request header specifies the URI which identifies a destination resource for methods such as COPY and MOVE, which take two URIs as parameters. Note that the absolute-URI production is defined in [RFC3986].

If the Destination value is an absolute URI, 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 with a 502 (Bad Gateway) response.

9.4. If Header

```
If = "If" ":" ( 1*No-tag-list | 1*Tagged-list)
No-tag-list = List
Tagged-list = Resource 1*List
Resource = Coded-URL
List = "(" 1*(["Not"](State-token | "[" entity-tag "]")) ")"
     ; No LWS allowed between "[", entity-tag and "]"
State-token = Coded-URL
```

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 is intended for use with any URI which represents state information, referred to as a state token, about a resource 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. The <DAV:no-lock> state token is an example of a state token that will never match an actual valid lock token. The purpose of this is described in Section 9.4.4.

The If header's purpose is to describe a series of state lists. If the state of the resource to which the header is applied does not match any of the specified state lists then the request MUST fail with a 412 (Precondition Failed). If one of the described state lists matches the state of the resource then the request may succeed.

The server MUST do authorization checks before checking this or any

conditional header. Assuming no other errors, the server MUST parse the If header when it appears on any request, evaluate all the clauses, and if the conditional evaluates to false, fail as described above.

9.4.1. No-tag-list Production

The No-tag-list production describes a series of state tokens and ETags. If multiple No-tag-list productions are used then one only needs to match the state of the resource for the method to be allowed to continue. All untagged tokens apply to the resource identified in the Request-URI.

Example - no-tag-list 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 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 being in the form (or (and <urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2> ["I am an ETag"]) (and ["I am another ETag"])).

9.4.2. Tagged-list Production

The tagged-list production may be used instead of the no-tag-list production, in order to scope each token to a specific resource. That is, it specifies that the lists following the resource specification only apply to the specified resource. The scope of the resource production begins with the list production immediately following the resource production and ends with the next resource production, if any. All clauses must be evaluated. If the state of the resource named in the tag does not match any of the associated state lists then the request MUST fail with a 412 (Precondition Failed).

The same URI MUST NOT appear more than once in a resource production in an If header.

9.4.3. Example - Tagged List If header in COPY

```
>>Request

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

That is the only success condition since the resource http://www.example.com/random never has the method applied to it (the only other resource listed in the If header) and http://www.example.com/resource2 is not listed in the If header.

9.4.4. Not Production

Every state token or ETag is either current, and hence describes the state of a resource, or is not current, and does not describe the state of a resource. The boolean operation of matching a state token or ETag to the current state of a resource thus resolves to a true or false value. The "Not" production is used to reverse that value. The scope of the not production is the state-token or entity-tag immediately following it.

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

When submitted with a request, this If header requires that all operand resources must not be locked with urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2 and must be locked with urn:uuid:58f202ac-22cf-11d1-b12d-002035b29092.

The Not production is particularly useful with a state token known not to ever identify a lock, such as the "<DAV:no-lock>" state token. The clause "Not <DAV:no-lock>" MUST evaluate to true. Thus, any "OR"

statement containing the clause "Not <DAV:no-lock>" MUST also evaluate to true.

9.4.5. Matching Function

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.

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.

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 or has a lock with a different token, the request MUST fail. If the 'specs' collection is locked (depth infinity) with that lock token, then this request could succeed, both because the If header evaluates to true, and because the lock token for the lock affecting the affected resource has been provided. Alternatively, a request where the 'rfc2518.txt' URL is associated with the lock token in the If header could also succeed.

9.4.6. 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 "Cache-Control: no-cache" request header MUST be used 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.

9.5. 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.

9.6. Overwrite Header

```
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 state of 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 the If-Match: * header of HTTP/1.1, 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.

9.7. Timeout Request 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.

Timeout response values MUST use a Second value or Infinite.

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.

The timeout counter MUST be restarted if a refresh LOCK request is successful. The timeout counter SHOULD NOT be restarted at any other time.

If the timeout expires then the lock may be lost. Specifically, if the server wishes to harvest the lock upon time-out, 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. Thus logs should be updated with the disposition of the lock, notifications should be sent, etc., just as they would be for an UNLOCK request.

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 off-line.

A client MUST NOT assume that just because the time-out has expired the lock has been lost. Likewise, a client MUST NOT assume that just because the time-out has not expired, the lock still exists (and for this reason, clients are strongly advised to use ETags as well).

10. Status Code Extensions to HTTP/1.1

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

10.1. 207 Multi-Status

The 207 (Multi-Status) status code provides status for multiple independent operations (see Section 12 for more information).

10.2. 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.

10.3. 423 Locked

The 423 (Locked) status code means the source or destination resource of a method is locked. This response SHOULD contain the 'lock-token-present' precondition element and corresponding 'href' in the error body.

10.4. 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).

10.5. 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 which 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.

11. 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 request MUST NOT be used when the server has created a new resource in response to the request.

11.1. 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 request contains a conditional header, and if that condition fails to hold, then this error code MUST be returned unless some other error is returned. On the other hand, if the client did not include a conditional header in the request, then the server MUST NOT use this error.

11.2. 414 Request-URI Too Long

This status code is used in HTTP 1.1 only for Request-URIs, because full URIs aren't used in other headers. WebDAV specifies full URLs in other headers, therefore this error MAY be used if the URI is too long in other locations as well.

12. Multi-Status Response

A Multi-Status response contains one 'response' element for each resource in the scope of the request (in no required order) or may be empty if no resources match the request. The default 207 (Multi-Status) response body is a text/xml or application/xml HTTP entity that contains a single XML element called 'multistatus', which contains a set of XML elements called response which 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. The 207 status code itself MUST NOT be considered a success response, it is only completely successful if all 'response' elements inside contain success status codes.

The body of a 207 Multi-Status response MUST contain a URL associated with each specific status code, so that the client can tell whether the error occurred with the source resource, destination resource or some other resource in the scope of the request.

12.1. 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.

12.2. URL Handling

A Multi-Status body contains one or more 'response' elements. Each response element describes a resource, and has an 'href' element identifying the resource. The 'href' element MUST contain an absolute URI or relative reference. It MUST NOT include "." or ".." as path elements.

If a 'href' element contains a relative reference, it MUST be resolved against the Request-URI. A relative reference MUST be an absolute path (note that clients are not known to support relative paths).

Identifiers for collections appearing in the results SHOULD end in a '/' character.

If a server allows resource names to include characters that aren't legal in HTTP URL paths, these characters must be percent-encoded on the wire (see [RFC3986], section 2.1). For example, it is illegal to use a space character or double-quote in a URI. URIs appearing in

PROPFIND or PROPPATCH XML bodies (or other XML marshalling defined in this specification) are still subject to all URI rules, including forbidden characters.

12.3. 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 13.9). 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.

12.4. Internal Status Codes

Section 8.3.1, Section 8.2.2, Section 8.7.1, Section 8.9.3 and Section 8.10.2 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.

13. XML Element Definitions

In this section, the final line of each section gives the element type declaration using the format defined in [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). The "Extensibility" field discusses how the element may be extended in the future (or in existing extensions to WebDAV.

All of the elements defined here may be extended by the addition of attributes and child elements not defined in this specification. All elements defined here are in the "DAV:" namespace.

13.1. activelock XML Element

Name: activelock

Purpose: Describes a lock on a resource.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

13.2. 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.

Extensibility: Normally empty, but MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

<!ELEMENT allprop EMPTY >

13.3. 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.

Extensibility: MAY be extended with child elements or attributes which SHOULD be ignored if not recognized.

```
<!ELEMENT collection EMPTY >
```

13.4. dead-props XML Element

Name: dead-props

Purpose: Specifies that all dead properties, names and values, should be returned in the response.

Extensibility: Normally empty, but MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

```
<!ELEMENT dead-props EMPTY >
```

13.5. depth XML Element

Name: depth

Purpose: The value of the Depth header.

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

Extensibility: MAY be extended with attributes which SHOULD be ignored.

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

13.6. error XML Element

Name: error

Purpose: Error responses, particularly 403 Forbidden and 409 Conflict, sometimes need more information to indicate what went wrong. When an error response contains a body in WebDAV, the body is in XML with the root element 'error'. The 'error' element SHOULD include an XML element with the code of a failed precondition or postcondition.

Description: Contains any XML element

Extensibility: Fully extensible with additional child elements, attributes or text (possibly mixed content). Unrecognized information items SHOULD be ignored if not recognized.

```
<!ELEMENT error ANY >
```

13.7. exclusive XML Element

Name: exclusive

Purpose: Specifies an exclusive lock

Extensibility: Normally empty, but MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

```
<!ELEMENT exclusive EMPTY >
```

13.8. href XML Element

Name: href

Purpose: Identifies the content of the element as a URI or a relative reference. 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: URI-reference (See section 4.1 of [RFC3986])

Extensibility: MAY be extended with attributes which SHOULD be ignored if not recognized.

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

13.9. 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 Multi-Status response, the 'location' element can be used to provide the

accompanying 'Location' header.

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

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

13.10. lockentry XML Element

Name: lockentry

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

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

13.11. 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.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

13.12. 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 contains a HTTP URL with the address of the root of the lock. The server SHOULD include this in all DAV: lockdiscovery property values and the response to LOCK requests.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

13.13. lockscope XML Element

Name: lockscope

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

Extensibility: SHOULD NOT be extended with child elements. MAY be extended with attributes which SHOULD be ignored.

```
<!ELEMENT lockscope (exclusive | shared) >
```

13.14. 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.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

13.15. 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.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

```
<!ELEMENT locktype (write) >
```

13.16. 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.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

```
<!ELEMENT multistatus (response+, responsedescription?)  >
```

13.17. owner XML Element

Name: owner

Purpose: Provides information about the principal taking out a lock.

Description Provides 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 of a homepage) who owns a lock. This information is provided by the client, and may only be altered by the server if the owner value provided by the client is empty.

Extensibility MAY be extended with child elements, mixed content, text content or attributes. Structured content, for example one or more 'href' child elements containing URIs of any kind, is RECOMMENDED.

```
<!ELEMENT owner ANY >
```

13.18. 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. No other elements may be used inside of a 'prop' element.

Extensibility MAY be extended with attributes which SHOULD be ignored if not recognized. Any child element of this element must be considered to be a property name, however these are not restricted to the property names defined in this document or other standards.

```
<!ELEMENT prop ANY >
```

13.19. 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. This XML element is multi-valued.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

```
<!ELEMENT propertyupdate (remove | set)+ >
```

13.20. 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', 'dead-props', 'allprop' and 'propname'. If 'prop' is used inside 'propfind' it MUST NOT contain property values.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized, as long as it still contains one of the required elements.

```
<!ELEMENT propfind ( propname | allprop | (prop, dead-props?) ) >
```

13.21. propname XML Element

Name: propname

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

Extensibility: Normally empty, but MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

```
<!ELEMENT propname EMPTY >
```

13.22. 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.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

13.23. remove XML element

Name: remove

Purpose: Lists the DAV 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.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

13.24. 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 a 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.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

13.25. responsedescription XML Element

Name: responsedescription

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

Description: This XML element provides either information suitable to be presented to a user (PCDATA) or a machine readable error code.

Extensibility: MAY be extended with additional attributes which SHOULD be ignored if not recognized.

```
<!ELEMENT responsedescription (#PCDATA | error) >
```

13.26. set XML element

Name: set

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

Description: The 'set' XML element MUST contain only a prop XML element. The elements contained by the prop XML element inside the 'set' XML 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.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

13.27. shared XML Element

Name: shared

Purpose: Specifies a shared lock

Extensibility: Normally empty, but MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

13.28. status XML Element

Name: status

Purpose: Holds a single HTTP status-line

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

Extensibility: MAY be extended with attributes which SHOULD be ignored.

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

13.29. timeout XML Element

Name: timeout

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

Value: TimeType (defined in Section 9.7).

Extensibility: MAY be extended with attributes which SHOULD be ignored.

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

13.30. write XML Element

Name: write

Purpose: Specifies a write lock.

Extensibility: Normally empty, but MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

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

14. 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 [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 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 or PROPPATCH requests.

A protected property is one which cannot be changed with a PROPPATCH request. There may be other requests which would result in a change to a protected property (as when a PUT request to an existing resource causes DAV:contentlength to change to a new value). 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 value could include LWS as defined in [RFC2616], section 4.2. Server implementors SHOULD NOT include extra LWS in these values, however client implementors MUST be prepared to handle extra LWS.

14.1. 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 behaviour: 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")

Extensibility: MAY contain attributes which SHOULD be ignored if not recognized.

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

14.2. 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 behaviour: This property value SHOULD be preserved in COPY and MOVE operations.

Description: The DAV:displayname property should be defined on all DAV compliant resources. If present, the property 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).

Extensibility: MAY contain attributes which SHOULD be ignored if not recognized.

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

14.3. 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 behaviour: 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.

Extensibility: MAY contain attributes which SHOULD be ignored if not recognized.

<!ELEMENT getcontentlanguage (#PCDATA) >

14.4. 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 behaviour: This property value is dependent on the size of the destination resource, not the value of the property on the source resource.

Extensibility: MAY contain attributes which SHOULD be ignored if not recognized.

<!ELEMENT getcontentlength (#PCDATA) >

14.5. 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: SHOULD NOT be protected, so clients may fix this value. Note that servers implementing RFC2518 might have made this a protected property as this is a new requirement.

COPY/MOVE behaviour: 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.

Extensibility: MAY contain attributes which SHOULD be ignored if not recognized.

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

14.6. 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 behaviour: This property value is dependent on the final state of the destination resource, not the value of the property on the source resource.

Description: The getetag property MUST be defined on any DAV compliant resource that returns the Etag header. Refer to RFC2616 for a complete definition of the semantics of an ETag. Note that changes in properties or lock state MUST not cause a resource's ETag to change.

Extensibility: MAY contain attributes which SHOULD be ignored if not recognized.

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

14.7. 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 behaviour: 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 is preserved in a MOVE even when the HTTP Last-Modified value SHOULD change. Thus, clients cannot rely on this value for caching and SHOULD use ETags.

Description: Note that 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.

Extensibility: MAY contain attributes which SHOULD be ignored if not recognized.

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

14.8. 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 behaviour: 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. If there are no locks, but the server supports locks, the property will be present but contain zero 'activelock' elements. If there is one or more lock, an 'activelock' element appears for each lock on the resource.

Extensibility: MAY be extended with additional child elements or attributes which SHOULD be ignored if not recognized.

```
<!ELEMENT lockdiscovery (activelock)* >
```

14.8.1. Example - Retrieving the lockdiscovery Property

>>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.

14.9. 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 behaviour: 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 13.3). 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.

Extensibility: MAY be extended with any child elements or attributes which SHOULD be ignored if not recognized.

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>
```

14.10. supportedlock Property

Name: supportedlock

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

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

COPY/MOVE behaviour: 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 which 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.

Extensibility: MAY be extended with any child elements or attributes which SHOULD be ignored if not recognized.

```
<!ELEMENT supportedlock (lockentry)* >
```

14.10.1. Example - Retrieving the DAV:supportedlock Property

```
>>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>
```

15. Precondition/postcondition XML elements

As introduced in section Section 8.1.5, 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 a DAV:error element in the appropriate DAV: responsedescription element. In all other error responses, the XML element MUST be returned as the child of a top-level DAV: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 DAV:error element MAY contain child elements with specific error information and MAY be extended with custom child elements and text (mixed content).

This mechanism does not take the place of using a correct numeric error 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 error. However, it does remove the need to define new numeric error codes. The machine-readable codes used for this purpose are XML elements classified as preconditions and postconditions, so naturally any group defining a new error code can use their own namespace. As always, the "DAV:" namespace is reserved for use by IETF-chartered WebDAV working groups.

A server supporting "bis" 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 error code and other supporting text, because even when clients do not automatically recognize error codes they can be quite useful in interoperability testing and debugging.

15.1. 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-present>
       <D:href>/workspace/webdav/</D:href>
     </D:lock-token-present>
   </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.

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.

```
<!ELEMENT no-external-entities EMPTY >
```

Name: lock-token-matches-request-uri

Use with: 400 Bad Request

Purpose: (precondition) -- A request may include a Lock-Token header to identify a lock for the purposes of an operation such as refresh LOCK or UNLOCK. However, if the Request-URI doe 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.

```
<!ELEMENT lock-token-matches-request-uri EMPTY >
```

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.

<!ELEMENT preserved-live-properties EMPTY >

Name: writable-property

Use with: 403 Forbidden

Purpose: (precondition) -- The client attempted to set a read-only property in a PROPPATCH (such as DAV:getetag).

<!ELEMENT writable-property EMPTY >

Name: propfind-finite-depth

Use with: 403 Forbidden

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

<!ELEMENT propfind-finite-depth EMPTY >

Name: lock-token-present

Use with: 4xx responses, e.g. 400 Bad Request or 423 Locked

Purpose: (precondition) -- The request could not succeed because a lock token should have been provided. 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 resonsible 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-present (href+) >

16. Instructions for Processing XML in DAV

All DAV compliant resources MUST ignore any unknown XML element and all its children encountered while processing a DAV method that uses XML as its command language.

This restriction also applies to the processing, by clients, of DAV property values where unknown 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 unknown 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.

Since XML can be transported as text/xml or application/xml, a DAV server MUST accept DAV method requests with XML parameters transported as either text/xml or application/xml, and a DAV client MUST accept XML responses using either text/xml or application/xml.

XML DTD fragments are included for all the XML elements defined in this specification. However, legal XML may not be valid according to any DTD due to namespace usage and extension rules, so the DTD is only informational. 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.

17. 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 are spoken of as being compliant, rather than servers. That is because theoretically some resources on a server could support different feature sets. E.g. a server could have a sub-repository where an advanced feature like server was supported, even if that feature was not supported on all servers.

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 compliant must also be class 1 compliant, and a resource that is compliant with "bis" must also be class 1 compliant.

17.1. 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.

17.2. 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 Time-Out 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.

17.3. Class 'bis'

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.

Example:

```
DAV: 1, bis
```

18. 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 encodings of the ISO 10646 multilingual plane. XML examples in this specification demonstrate use of the charset parameter of the Content-Type header, as defined in [RFC3023], as well as the XML declarations which provide charset identification information for MIME and XML processors.

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 [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 USASCII 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.

19. 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.

19.1. 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 Basic authentication credentials in a WWW-Authenticate header unless the connection is secure. Examples of secure connections include a Transport Layer Security (TLS) connection employing a strong cipher suite with mutual authentication of client and server, or a connection over a network which is physically secure, for example, an isolated network in a building with restricted access.

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 which is useful in a wide range of scenarios.

19.2. 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.

- The underlying storage can be attacked by PUTting extremely large files.
- Asking for recursive operations on large collections can attack processing time.
- 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).

19.3. 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.

19.4. 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.

19.5. 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.

19.6. Implications of XML Entities

XML supports a facility known as "external entities", defined in section 4.2.2 of [XML], which instruct 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 implementor chooses not to handle external XML entities, it SHOULD respond to requests containing external entities with the precondition defined above (no-external-entities).

There is also the scalability risk that would accompany a widely deployed application which 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 which 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 [XML]. A small, carefully crafted request using nested internal entities may require enormous amounts of memory and/or processing time to process. Server implementors 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.

19.7. Risks Connected with Lock Tokens

This specification encourages the use of "A Universally Unique Identifier (UUID) URN Namespace" ([RFC4122]) for lock tokens Section 6.3, in order to guarantee their uniqueness across space and time. Variant 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".

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

- It is possible to track the movement of hardware from subnet to subnet.
- It may be possible to identify the manufacturer of the hardware running a WebDAV server.
- 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 involve the host address and therefore do not suffer from this risk.

19.8. Hosting malicious scripts executed on client machines

HTTP has the ability to host scripts which are executed on client machines. These scripts can be used to read another user's cookies and therefore may provide an attacker the ability to use another user's session, assume their identity temporarily and gain access to their resources. Other attacks are also possible with client-executed scripts.

WebDAV may worsen this situation only by making it easier for a Web server to host content provided by many different authors (making it harder to trust the content providers) and to host content with restricted access alongside public pages (see particularly RFC3744).

HTTP servers may mitigate some of these threats by filtering content in areas where many authors contribute pages -- the server could, for example, remove script from HTML pages.

This vulnerability should provide yet another reason for server implementors and administrators not to replace authentication mechanisms with cookie-based session tokens if there's any sensitive

information relying on the authenticated identity.

HTTP and WebDAV client implementors might consider locking down the use of scripts and cookies based on these considerations.

20. IANA Considerations

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.

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. 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 RFC2518

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, both in helping the formation of the working group and in patiently coaching the authors along the way. In so many ways he has set high standards we have toiled to meet. The contributions of Judith Slein in clarifying the requirements, and in patiently reviewing draft after draft, 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 RFC2518 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 Contributors to This Specification

Valuable contributions to this specification came from some already named. New and significant contributors to this specification must also be gratefully acknowledged. Julian Reschke, Geoff Clemm, Joel Soderberg, and Dan Brotsky hashed 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.

Jason Crawford tracked issue status for this document for a period of years, followed by Elias Sinderson. Elias and Jim Whitehead collaborated on specific document text.

21.1. Previous Authors' Addresses

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.

22. References

22.1. Normative References

[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. 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.

[W3C.REC-xml-names-19990114] Hollander, D., Bray, T., and A. Layman, "Namespaces in XML", W3C REC REC-xml-names-19990114, January 1999.

[XML] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Third Edition)", W3C REC-xml-20040204, February 2004, <http://www.w3.org/TR/2004/REC-xml-20040204>.

22.2. Informational 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.

[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.

[W3C.REC-xml-infoset-20040204]
Tobin, R. and J. Cowan, "XML Information Set (Second Edition)", W3C REC REC-xml-infoset-20040204, February 2004.

Appendix A. Notes on Processing XML Elements

A.1. 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.

A.2. 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 white space, 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.

A.3. 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 - Unknown 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 16, it must ignore it. 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

The PUT and DELETE methods are defined in HTTP and thus may be used by HTTP clients, but the responses to PUT and DELETE have been extended in this specification, so some special consideration on backward compatibility is worthwhile.

First, if a PUT or DELETE request includes a header defined in this specification (Depth or If), the server can assume the request comes from a WebDAV-compatible client. The server may even be able to track a number of requests across a session and know that a client is a WebDAV client. However, this kind of detection may not be necessary.

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 a 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 a 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. Thus, a server MAY choose to 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) depending on what the error was. This approach would 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 may be more theoretical than practical. Thus, servers MAY instead choose to treat any such DELETE request as a WebDAV request, and send a 207 Multistatus containing more detail about what resources could not be deleted.

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 linear white
  ; space (LWS) is not allowed between elements of this production.

Extension = path
   ; path is defined in section 3.3 of RFC3986
```

Appendix D. Guidance for Clients Desiring to Authenticate

Many WebDAV clients already 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 the client is authenticated or not -- 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 do 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 8.1.3), 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 RFC2617:

"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."

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 E. Summary of changes from RFC2518

This section describes changes that are likely to result in implementation changes due to tightened requirements or changed behavior.

E.1. Changes Notable to Server Implementors

Tightened requirements for storing property values (Section 4.4) when "xml:lang" appears and also when values are XML fragments (specifically on preserving prefixes, namespaces and whitespace.)

Tightened requirements on which properties are protected and computed (Section 14).

Several tightened requirements for general response handling (Section 8.1), including response bodies for use with errors, ETag and Location header, and reminder to use Date header.

Tightened requirements for URL construction in PROPFIND (Section 8.2) responses.

Tightened requirements for checking identity of lock owners (Section 7.1) during operations affected by locks.

Tightened requirements for copying properties (Section 8.9.2) and moving properties (Section 8.10.1).

Tightened requirements on preserving owner field in locks (Section 8.11). Added "lockroot" element to lockdiscovery information.

New value for "DAV:" header (Section 9.1) to advertise support for this specification.

Tightened requirement for "Destination:" header (Section 9.3) to work with path values

Some changes for "If:" header (Section 9.4), including "DAV:no-lock" state token and requirement to evaluate entire header.

Support for UTF-16 now required (ref (Section 18)).

Removed definition of "source" property and special handling for dynamic resources

Replaced lock-null resources with simpler locked empty resources (Section 7.6). Lock-null resources are now not compliant with the

requirements in this specification.

Encouraged servers to change ETags (Section 8.1.4) only when body of resource changes.

The definition of the 102 Processing response was removed and servers ought to stop sending that response when they implement this specification.

Previously, servers were encouraged to return 409 status code in response to a collection LOCK request if some resource could not be locked. Now servers should use 207 Multi-Status instead.

Only 'rfc1123-date' productions are legal as values for DAV: getlastmodified.

New explicit requirement to do authorization checks before conditional checks (Section 8.1.3).

E.2. Changes Notable to Client Implementors

Tightened requirements for supporting WebDAV collections (Section 5.2) within resources that do not support WebDAV (e.g. servlet containers).

Redefined 'allprop' PROPFIND requests so that the server does not have to return all properties.

Required to handle empty multistatus responses in PROPFIND responses (Section 8.2)

No more "propertybehavior" specification allowed in MOVE and COPY requests

The change in behavior of LOCK with an unmapped URL might affect client implementations that counted on lock-null resources disappearing when the lock expired. Clients can no longer rely on that cleanup happening.

Support for UTF-16 now required (ref (Section 18)).

Removed definition of "source" property and special handling for dynamic resources.

The definition of the 102 Processing response was removed and clients can safely remove code (if any) that deals with this.

Servers may now reject PROPFIND depth "infinity" requests.

Clients use Lock-Token header, not the If header, to provide lock token when renewing lock. Section 8.11.1

Appendix F. Change Log (to be removed by RFC Editor before publication)

F.1. Changes from -05 to -06

Specified that a successful LOCK request to an unmapped URL creates a new, empty locked resource.

Resolved UNLOCK_NEEDS_IF_HEADER by clarifying that only Lock-Token header is needed on UNLOCK.

Added Section 15 on preconditions and postconditions and defined a number of preconditions and postconditions. The 'lock-token-present' precondition resolves the REPORT_OTHER_RESOURCE_LOCKED issue.

Added example of matching lock token to URI in the case of a collection lock in the If header section.

Removed ability for Destination header to take "abs_path" in order to keep consistent with other places where client provides URLs (If header, href element in request body)

Clarified the href element - that it generally contains HTTP URIs but not always.

Attempted to fix the BNF describing the If header to allow commas

Clarified presence of Depth header on LOCK refresh requests.

F.2. Changes in -07

Added text to "COPY and the Overwrite Header" section to resolve issue OVERWRITE_DELETE_ALL_TOO_STRONG.

Added text to "HTTP URL Namespace Model" section to provide more clarification and examples on what consistency means and what is not required, to resolve issue CONSISTENCY.

Resolve DEFINE_PRINCIPAL by importing definition of principal from RFC3744.

Resolve INTEROP_DELETE_AND_MULTISTATUS by adding appendix 3 discussing backward-compatibility concerns.

Resolve DATE_FORMAT_GETLASTMODIFIED by allowing only rfc1123-date, not HTTP-date for getlastmodified.

Resolve COPY_INTO_YOURSELF_CLARIFY by adding sentence to first para. of COPY section.

Confirm that WHEN_TO_MULTISTATUS_FOR_DELETE_1 and WHEN_TO_MULTISTATUS_FOR_DELETE_2 are resolved and tweak language in DELETE section slightly to be clearly consistent.

More text clarifications to deal with several of the issues in LOCK_ISSUES. This may not completely resolve that set but we need feedback from the originator of the issues at this point.

Resolved COPY_INTO_YOURSELF_CLARIFY with new sentence in Copy For Collections section.

Double checked that LEVEL_OR_CLASS is resolved by using class, not level.

Further work to resolve IF_AND_AUTH and LOCK_SEMANTICS, clarifying text on using locks and being authenticated.

Added notes on use of 503 status response to resolve issue PROPFIND_INFINITY

Removed section on other uses of Metadata (and associated references)

Added reference to RFC4122 for lock tokens and removed section on generating UUIDs

Explained that even with language variation, a property has only one value (section 4.5).

Added section on lock owner (7.1) and what to do if lock requested by unauthenticated user

Removed section 4.2 -- justification on why to have metadata, not needed now

Removed paragraph in section 5.2 about collections with resource type "DAV:collection" but which are non-WebDAV compliant -- not implemented.

F.3. Changes in -08

Added security considerations section on scripts and cookie sessions, suggested by Barry Lind

Clarified which error codes are defined and undefined in MultiStatus

Moved opaquelocktoken definition to an appendix and refer to RFC4122 for use of 'urn:uuid:' URI scheme; fix all lock token examples to use this.

Multi-status responses contain URLs which MUST either be absolute (and begin with the Request-URI or MUST be relative with new limitations. (bug 12)

Moved status code sections before example sections within PROPFIND section for section ordering consistency.

Clarified use of Location header with Multi-Status

Bugzilla issue resolutions: bugs 9, 12, 14, 19, 20, 29, 30, 34, 36, 102 and 172.

F.4. Changes in -09

Bugzilla editorial issues: bugs 30, 57, 63, 68, 88, 89, 168, 180, 182, 185, 187.

More clarity between URL namespaces and XML namespaces, particularly at the beginning of paragraphs using the word namespace

More consistency in referring to properties with the namespace, as in "DAV:lockdiscovery", and referring to XML element names in single quotes, e.g. 'allprop' element.

Figure (example) formatting fixes

Bugzilla issues: bugs 24, 37, 39, 43, 45, 27, 25

Replaced references to "non-null state" of resources with more clear language about URLs that are mapped to resources, bug 25. Also added definition of URL/URI mapping. Bug 40.

Bugzilla issues: bug 7, 8, 9, 41, 47, 51, 62, 93, 171, 172. Bugs 28 and 94 were iterated on.

Bugzilla issues: 56, 59, 79, 99, 103, 175, 178. Part of bug 23. Iteration on bug 10.

Iteration on bugs 10, 46 and 47. Bug 11.

Remove "102 Processing" response

Fix bug 46, 105, 107, 120, 140 and 201.

Another stab at bug 12 - relative v. absolute URLs in Multi-Status response hrefs

Fix bug 6, 11, 15, 16, 28, 32, 42, 51, 52, 53, 58, 60, 62, 186, 189,

191, 199, 200

Fix bug 96

F.5. Chandles in -10

Clarify lock intro text on when a client might use another client's lock token - suggestion by Geoff, Nov 15

Removed Force-Authenticate header and instead added an appendix explaining how existing mechanisms might resolve the need of clients to get an authentication challenge (bug 18).

Bug 62, 113, 125, 131, 143, 144, 171, 193

Bug 176, 177, 179, 181, 184, 206, 207, 208

Author's Address

Lisa Dusseault
Open Source Application Foundation
2064 Edgewood Dr.
Palo Alto, CA 94303
US

Email: lisa@osafoundation.org

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.