Network Working Group
Request for Comments: 2518
Category: Standards Track

Y. Goland
Microsoft
E. Whitehead
UC Irvine
A. Faizi
Netscape
Novell
D. Jensen
Novell

February 1999

HTTP Extensions for Distributed Authoring -- WEBDAV

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This document specifies 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, namespace manipulation, and resource locking (collision avoidance).

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.

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].

The sections below provide a detailed introduction to resource properties (section 4), collections of resources (section 5), and locking operations (section 6). These sections introduce the abstractions manipulated by the WebDAV-specific HTTP methods described in section 8, "HTTP Methods for Distributed Authoring".

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 Extensible Markup Language (XML) [REC-XML] request entity body, or in an HTTP header. The use of XML to encode method parameters was motivated by the ability to add extra XML elements to existing structures, providing extensibility; and by XML's ability to encode information in ISO 10646 character sets, providing internationalization support. As a rule of thumb, parameters are encoded in XML entity bodies when they have unbounded length, or when they may be shown to a human user and hence require encoding in an ISO 10646 character set. Otherwise, parameters are encoded within HTTP headers. Section 9 describes the new HTTP headers used with WebDAV methods.

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.

XML elements used in this specification are defined in section 12.

The XML namespace extension (Appendix 4) 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.

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. New status codes developed for the WebDAV methods are defined in section 10. Since some WebDAV methods may operate over many resources, the Multi-Status response has been introduced to return status information for multiple resources. The Multi-Status response is described in section 11.

WebDAV employs the property mechanism to store information about the current state of the resource. For example, when a lock is taken out on a resource, a lock information property describes the current state of the lock. Section 13 defines the properties used within the WebDAV specification.

Finishing off the specification are sections on what it means to be compliant with this specification (section 15), on internationalization support (section 16), and on security (section 17).

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 [RFC2068]. Since this augmented BNF uses the basic production rules provided in section 2.2 of [RFC2068], 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 RFC 2119 [RFC2119].

3 Terminology

URI/URL - A Uniform Resource Identifier and Uniform Resource Locator, respectively. These terms (and the distinction between them) are defined in [RFC2396].

Collection - A resource that contains a set of URIs, termed member URIs, which identify member resources and meets the requirements in section 5 of this specification.

Member URI - A URI which is a member of the set of URIs contained by a collection.

Internal Member URI - A Member URI that is immediately relative to the URI of the collection (the definition of immediately relative is given in 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 “getcontentlength” property 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.

Null Resource - A resource which responds with a 404 (Not Found) to any HTTP/1.1 or DAV method except for PUT, MKCOL, OPTIONS and LOCK. A NULL resource MUST NOT appear as a member of its parent collection.

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 Existing Metadata Proposals

Properties have long played an essential role in the maintenance of large document repositories, and many current proposals contain some notion of a property, or discuss web metadata more generally. These include PICS [REC-PICS], PICS-NG, XML, Web Collections, and several proposals on representing relationships within HTML. Work on PICS-NG and Web Collections has been subsumed by the Resource Description Framework (RDF) metadata activity of the World Wide Web Consortium. RDF consists of a network-based data model and an XML representation of that model.

Some proposals come from a digital library perspective. These include the Dublin Core [RFC2413] metadata set and the Warwick Framework [WF], a container architecture for different metadata schemas. The literature includes many examples of metadata, including MARC [USMARC], a bibliographic metadata format, and a technical report bibliographic format employed by the Dienst system [RFC1807]. Additionally, the proceedings from the first IEEE Metadata conference describe many community-specific metadata sets.

Participants of the 1996 Metadata II Workshop in Warwick, UK [WF], noted that “new metadata sets will develop as the networked infrastructure matures” and “different communities will propose, design, and be responsible for different types of metadata.” These observations can be corroborated by noting that many community-specific sets of metadata already exist, and there is significant motivation for the development of new forms of metadata as many communities increasingly make their data available in digital form, requiring a metadata format to assist data location and cataloging.

4.3 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.

4.4 Property Values

The value of a property when expressed in XML MUST be well formed.

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 will 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.

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 [RFC2396], 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 Media Independent Links

Although HTML resources support links to other resources, the Web needs more general support for links between resources of any media type (media types are also known as MIME types, or content types). WebDAV provides such links. A WebDAV link is a special type of property value, formally defined in section 12.4, that allows typed connections to be established between resources of any media type. The property value consists of source and destination Uniform Resource Identifiers (URIs); the property name identifies the link type.

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.

Neither HTTP/1.1 nor WebDAV require that the entire HTTP URL namespace be consistent. However, certain WebDAV methods are prohibited from producing results that cause namespace inconsistencies.

Although implicit in [RFC2068] and [RFC2396], 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 URIs and a set of properties, but which may have additional state such as entity bodies returned by GET. An internal member URI MUST be immediately relative to a base URI of the collection. That is, the internal member URI is equal to a containing collection's URI 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 [RFC2396].

Any given internal member URI MUST only belong to the collection once, i.e., it is illegal to have multiple instances of the same URI 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 URIs U and V, for which U is immediately relative to V, B MUST be a collection that has U as an internal member URI. So, if the resource with URL http://foo.com/bar/blah is WebDAV compliant and if the resource with URL http://foo.com/bar/ is WebDAV compliant then the resource with URL http://foo.com/bar/ must be a collection and must contain URL http://foo.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://foo.com/bar/blah is not WebDAV compliant and the URL http://foo.com/bar/ identifies a collection then URL http://foo.com/bar/blah may or may not be an internal member of the collection with URL http://foo.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 trailing slash is automatically appended. Due to this, a resource may accept a URI without a trailing "/" to point to a collection. In this case it SHOULD return a content-location header in the response pointing to the URI ending with the "/". For example, if a client invokes a method on http://foo.bar/blah (no trailing slash), the resource http://foo.bar/blah/ (trailing slash) may respond as if the operation were invoked on it, and should return a content-location header with http://foo.bar/blah/ in it. In general clients SHOULD use the "/" form of collection names.

A resource MAY be a collection but not be WebDAV compliant. That is, the resource may comply with all the rules set out in this specification regarding how a collection is to behave without necessarily supporting all methods that a WebDAV compliant resource is required to support. In such a case the resource may return the DAV:resourcetype property with the value DAV:collection but MUST NOT return a DAV header containing the value "1" on an OPTIONS response.

5.3 Creation and Retrieval of Collection Resources

This document specifies the MKCOL method to create new collection resources, rather than using the existing HTTP/1.1 PUT or POST method, for the following reasons:

In HTTP/1.1, the PUT method is defined to store the request body at the location specified by the Request-URI. While a description format for a collection can readily be constructed for use with PUT, the implications of sending such a description to the server are undesirable. For example, if a description of a collection that omitted some existing resources were PUT to a server, this might be interpreted as a command to remove those members. This would extend PUT to perform DELETE functionality, which is undesirable since it changes the semantics of PUT, and makes it difficult to control DELETE functionality with an access control scheme based on methods.

While the POST method is sufficiently open-ended that a “create a collection” POST command could be constructed, this is undesirable because it would be difficult to separate access control for collection creation from other uses of POST.

The exact definition of the behavior of GET and PUT on collections is defined later in this document.

5.4 Source Resources and Output Resources

For many resources, the entity returned by a GET method exactly matches the persistent state of the resource, for example, a GIF file stored on a disk. For this simple case, the URI at which a resource is accessed is identical to the URI at which the source (the persistent state) of the resource is accessed. This is also the case for HTML source files that are not processed by the server prior to transmission.

However, the server can sometimes process HTML resources before they are transmitted as a return entity body. For example, a server-side-include directive within an HTML file might instruct a server to replace the directive with another value, such as the current date. In this case, what is returned by GET (HTML plus date) differs from the persistent state of the resource (HTML plus directive). Typically there is no way to access the HTML resource containing the unprocessed directive.

Sometimes the entity returned by GET is the output of a data-producing process that is described by one or more source resources (that may not even have a location in the URI namespace). A single data-producing process may dynamically generate the state of a potentially large number of output resources. An example of this is a CGI script that describes a “finger” gateway process that maps part of the namespace of a server into finger requests, such as http://www.foo.bar.org/finger_gateway/user@host.

In the absence of distributed authoring capabilities, it is acceptable to have no mapping of source resource(s) to the URI namespace. In fact, preventing access to the source resource(s) has desirable security benefits. However, if remote editing of the source resource(s) is desired, the source resource(s) should be given a location in the URI namespace. This source location should not be one of the locations at which the generated output is retrievable, since in general it is impossible for the server to differentiate requests for source resources from requests for process output resources. There is often a many-to-many relationship between source resources and output resources.

On WebDAV compliant servers the URI of the source resource(s) may be stored in a link on the output resource with type DAV:source (see section 13.10 for a description of the source link property). Storing the source URIs in links on the output resources places the burden of discovering the source on the authoring client. Note that the value of a source link is not guaranteed to point to the correct source. Source links may break or incorrect values may be entered. Also note that not all servers will allow the client to set the source link value. For example a server which generates source links on the fly for its CGI files will most likely not allow a client to set the source link value.

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. This is a lock where the access right in question is only granted to a single principal. The need for this arbitration results from a desire to avoid having to merge results.

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 get 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 server is not required to support locking in any form. If the server 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. A lock token is returned by every successful LOCK operation in the lockdiscovery property in the response body, and can also be found through lock discovery on a resource.

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.

This specification provides a lock token URI scheme called opaquelocktoken that meets the uniqueness requirements. However resources are free to return any URI scheme so long as it meets the uniqueness requirements.

Having a lock token provides no special access rights. Anyone can find out anyone else's lock token by performing lock discovery. Locks MUST be enforced based upon whatever authentication mechanism is used by the server, not based on the secrecy of the token values.

6.4 opaquelocktoken Lock Token URI Scheme

The opaquelocktoken URI scheme is designed to be unique across all resources for all time. Due to this uniqueness quality, a client may submit an opaque lock token in an If header on a resource other than the one that returned it.

All resources MUST recognize the opaquelocktoken scheme and, at minimum, recognize that the lock token does not refer to an outstanding lock on the resource.

In order to guarantee uniqueness across all resources for all time the opaquelocktoken requires the use of the Universal Unique Identifier (UUID) mechanism, as described in [ISO-11578].

Opaquelocktoken generators, however, have a choice of how they create these tokens. They can either generate a new UUID for every lock token they create or they can create a single UUID and then add extension characters. If the second method is selected then the program generating the extensions MUST guarantee that the same extension will never be used twice with the associated UUID.

```
OpaqueLockToken-URI = "opaquelocktoken:" UUID [Extension]  ; The UUID
production is the string representation of a UUID, as defined in [ISO-
11578]. Note that white space (LWS) is not allowed between elements of
this production.

Extension = path  ; path is defined in section 3.2.1 of RFC 2068 [RFC2068]
```

6.4.1 Node Field Generation Without the IEEE 802 Address

UUIDs, as defined in [ISO-11578], contain a “node” field that contains one of the IEEE 802 addresses for the server machine. As noted in section 17.8, there are several security risks associated with exposing a machine's IEEE 802 address. This section provides an alternate mechanism for generating the “node” field of a UUID which does not employ an IEEE 802 address. WebDAV servers MAY use this algorithm for creating the node field when generating UUIDs. The text in this section is originally from an Internet-Draft by Paul Leach and Rich Salz, who are noted here to properly attribute their work.

The ideal solution is to obtain a 47 bit cryptographic quality random number, and use it as the low 47 bits of the node ID, with the most significant bit of the first octet of the node ID set to 1. This bit

is the unicast/multicast bit, which will never be set in IEEE 802 addresses obtained from network cards; hence, there can never be a conflict between UUIDs generated by machines with and without network cards.

If a system does not have a primitive to generate cryptographic quality random numbers, then in most systems there are usually a fairly large number of sources of randomness available from which one can be generated. Such sources are system specific, but often include:

- the percent of memory in use
- the size of main memory in bytes
- the amount of free main memory in bytes
- the size of the paging or swap file in bytes
- free bytes of paging or swap file
- the total size of user virtual address space in bytes
- the total available user address space bytes
- the size of boot disk drive in bytes
- the free disk space on boot drive in bytes
- the current time
- the amount of time since the system booted
- the individual sizes of files in various system directories
- the creation, last read, and modification times of files in various system directories
- the utilization factors of various system resources (heap, etc.)
- current mouse cursor position
- current caret position
- current number of running processes, threads
- handles or IDs of the desktop window and the active window
- the value of stack pointer of the caller
- the process and thread ID of caller
- various processor architecture specific performance counters (instructions executed, cache misses, TLB misses)

(Note that it is precisely the above kinds of sources of randomness that are used to seed cryptographic quality random number generators on systems without special hardware for their construction.)

In addition, items such as the computer's name and the name of the operating system, while not strictly speaking random, will help differentiate the results from those obtained by other systems.

The exact algorithm to generate a node ID using these data is system specific, because both the data available and the functions to obtain them are often very system specific. However, assuming that one can concatenate all the values from the randomness sources into a buffer, and that a cryptographic hash function such as MD5 is available, then any 6 bytes of the MD5 hash of the buffer, with the multicast bit (the high bit of the first byte) set will be an appropriately random node ID.

Other hash functions, such as SHA-1, can also be used. The only requirement is that the result be suitably random _ in the sense that the outputs from a set uniformly distributed inputs are themselves uniformly distributed, and that a single bit change in the input can be expected to cause half of the output bits to change.

6.5 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. Lock capability discovery differs from discovery of supported access control types, since there may be access control types without corresponding lock types. A client can determine what lock types the server supports by retrieving the supportedlock property.

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

6.6 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 lockdiscovery property is provided. This property lists all outstanding locks, describes their type, and where available, provides their lock token.

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

6.7 Usage Considerations

Although the locking mechanisms specified here 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 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.

7.1 Methods Restricted by Write Locks

A write lock MUST prevent a principal without the lock from successfully executing a PUT, POST, PROPPATCH, LOCK, UNLOCK, MOVE, DELETE, or MKCOL on the locked resource. All other current methods, GET in particular, function independently of the lock.

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

7.2 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.

7.3 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.

7.4 Write Locks and Null Resources

It is possible to assert a write lock on a null resource in order to lock the name.

A write locked null resource, referred to as a lock-null resource, MUST respond with a 404 (Not Found) or 405 (Method Not Allowed) to any HTTP/1.1 or DAV methods except for PUT, MKCOL, OPTIONS, PROPFIND, LOCK, and UNLOCK. A lock-null resource MUST appear as a member of its parent collection. Additionally the lock-null resource MUST have defined on it all mandatory DAV properties. Most of these properties, such as all the get* properties, will have no value as a lock-null resource does not support the GET method. Lock-Null resources MUST have defined values for lockdiscovery and supportedlock properties.

Until a method such as PUT or MKCOL is successfully executed on the lock-null resource the resource MUST stay in the lock-null state. However, once a PUT or MKCOL is successfully executed on a lock-null resource the resource ceases to be in the lock-null state.

If the resource is unlocked, for any reason, without a PUT, MKCOL, or similar method having been successfully executed upon it then the resource MUST return to the null state.

7.5 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 URIs of the collection by non-lock owners. As a consequence, when a principal issues a PUT or POST request to create a new resource under a URI which needs to be an internal member of a write locked collection to maintain HTTP namespace consistency, or issues a DELETE to remove a resource which has a URI which is an existing internal member URI of a write locked collection, this request MUST fail if the principal does not have a write lock on the collection.

However, if a write lock request is issued to a collection containing member URIs 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.

If a lock owner causes the URI of a resource to be added as an internal member URI of a 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.

7.6 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 in the If header for all locked resources that a method may interact with or the method MUST fail. For example, if a resource is to be moved and both the source and destination are locked then two lock tokens must be submitted, one for the source and the other for the destination.

7.6.1 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>
    (<opaquelocktoken: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.

7.7 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, the resource is subject to being added to an existing lock at the destination, as specified in section 7.5. 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 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 7.6, an If header must be submitted containing a lock token for both the source and destination.

7.8 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.

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

8 HTTP Methods for Distributed Authoring

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 [REC-XML]. All XML used in either requests or responses MUST be, at minimum, well formed. If a server receives ill-formed XML in a request it MUST reject the entire request with a 400 (Bad Request). If a client receives ill-formed XML in a response then it MUST NOT assume anything about the outcome of the executed method and SHOULD treat the server as malfunctioning.

8.1 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 URIs. All DAV compliant resources MUST support the PROPFIND method and the propfind XML element (section 12.14) 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 with internal member URIs. DAV compliant servers MUST support the “0”, “1” and “infinity” behaviors. 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, all property values, or a list of the names of the resource’s properties. A client may choose not to submit a request body. An empty PROPFIND request body MUST be treated as a request for the names and values of all properties.

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 URIs MUST include a response XML element for each member URI of the collection, to whatever depth was requested. Each response XML element MUST contain an href XML element that gives the URI 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 URIs are returned as a flat list whose order of entries is not significant.

In the case of allprop and propname, if a principal does not have the right to know whether a particular property exists then the property should be silently excluded from the response.

The results of this method SHOULD NOT be cached.

8.1.1 Example - Retrieving Named Properties

>>Request

```
PROPFIND  /file HTTP/1.1
Host: www.foo.bar
Content-type: text/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.foo.bar/boxschema/">
            <R:bigbox/>
            <R:author/>
            <R:DingALing/>
            <R:Random/>
      </D:prop>
</D:propfind>
```

>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/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.foo.bar/file</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>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.foo.bar/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.

8.1.2 Example - Using allprop to Retrieve All Properties

>>Request

```
PROPFIND  /container/ HTTP/1.1
Host: www.foo.bar
Depth: 1
Content-Type: text/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: text/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.foo.bar/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>http://www.foo.bar/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 properties defined on each resource.

The resource http://www.foo.bar/container/ has six properties defined on it:

http://www.foo.bar/boxschema/bigbox, http://www.foo.bar/boxschema/author, DAV:creationdate, DAV:displayname, DAV:resourcetype, and DAV:supportedlock.

The last four properties are WebDAV-specific, defined in section 13. Since GET is not supported on this resource, the get* properties (e.g., getcontentlength) are not defined on this resource. The DAV-specific properties assert that “container” was created on December 1, 1997, at 5:42:21PM, in a time zone 8 hours west of GMT (creationdate), has a name of “Example collection” (displayname), a collection resource type (resourcetype), and supports exclusive write and shared write locks (supportedlock).

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

http://www.foo.bar/boxschema/bigbox (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 (creationdate), has a name of “Example HTML resource” (displayname), a content length of 4525 bytes (getcontentlength), a MIME type of “text/html” (getcontenttype), an entity tag of “zzyzx” (getetag), was last modified on Monday, January 12, 1998, at 09:25:56 GMT (getlastmodified), has an empty resource type, meaning that it is not a collection (resourcetype), and supports both exclusive write and shared write locks (supportedlock).

8.1.3 Example - Using propname to Retrieve all Property Names

>>Request

```
PROPFIND  /container/ HTTP/1.1
Host: www.foo.bar
Content-Type: text/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: text/xml; charset="utf-8"
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<multistatus xmlns="DAV:">
      <response>
            <href>http://www.foo.bar/container/</href>
            <propstat>
                  <prop xmlns:R="http://www.foo.bar/boxschema/">
                        <R:bigbox/>
                        <R:author/>
                        <creationdate/>
                        <displayname/>
                        <resourcetype/>
                        <supportedlock/>
                  </prop>
                  <status>HTTP/1.1 200 OK</status>
            </propstat>
      </response>
      <response>
            <href>http://www.foo.bar/container/front.html</href>
            <propstat>
                  <prop xmlns:R="http://www.foo.bar/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.foo.bar/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 progeny should be returned.

Consistent with the previous example, resource http://www.foo.bar/container/ has six properties defined on it, http://www.foo.bar/boxschema/bigbox, http://www.foo.bar/boxschema/author, DAV:creationdate, DAV:displayname, DAV:resourcetype, and DAV:supportedlock.

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

This example also demonstrates the use of XML namespace scoping, and the default namespace. Since the "xmlns" attribute does not contain an explicit "shorthand name" (prefix) letter, 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 schema.

8.2 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 of the DAV schema. 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 the order instructions are received (i.e., from top to bottom). 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 12.13.

8.2.1 Status Codes for use with 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 command succeeded. As there can be a mixture of sets and removes in a body, a 201 (Created) seems inappropriate.

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

409 (Conflict) - The client has provided a value whose semantics are not appropriate for the property. This includes trying to set read-only properties.

423 (Locked) - The specified resource is locked and the client either is not a lock owner or the lock type requires a lock token to be submitted and the client did not submit it.

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

8.2.2 Example - PROPPATCH

>>Request

```
PROPPATCH /bar.html HTTP/1.1
Host: www.foo.com
Content-Type: text/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: text/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.foo.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 http://www.w3.com/standards/z39.50/Authors property, and to remove the property http://www.w3.com/standards/z39.50/Copyright-Owner. 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.

8.3 MKCOL Method

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

8.3.1 Request

MKCOL creates a new collection resource at the location specified by the Request-URI. If the resource identified by the Request-URI is non-null 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 neither /a/b/ nor /a/b/c/ exists, 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 behavior of a MKCOL request when the body is present is 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. The exact behavior of MKCOL for various request media types is undefined in this document, and will be specified in separate documents.

8.3.2 Status Codes

Responses from a MKCOL request MUST NOT be cached as MKCOL has non-idempotent semantics.

201 (Created) - The collection or structured resource was created in its entirety.

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 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 a deleted/non-existent resource.

409 (Conflict) - A collection cannot be made at the Request-URI until one or more intermediate collections have been created.

415 (Unsupported Media Type)- The server does not support the request type of the body.

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

8.3.3 Example - MKCOL

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

>>Request

```
MKCOL /webdisc/xfiles/ HTTP/1.1
Host: www.server.org
```

>>Response

```
HTTP/1.1 201 Created
```

8.4 GET, HEAD for Collections

The semantics of GET are unchanged when applied to a collection, since GET is defined as, "retrieve whatever information (in the form of an entity) is identified by the Request-URI" [RFC2068]. 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.

8.5 POST for Collections

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

8.6 DELETE

8.6.1 DELETE for Non-Collection Resources

If the DELETE method is issued to a non-collection resource whose URIs are an internal member of one or more collections, then during DELETE processing a server MUST remove any URI for the resource identified by the Request-URI from collections which contain it as a member.

8.6.2 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 URIs are to be deleted.

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

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). 424 (Failed Dependency) errors SHOULD NOT be in the 207 (Multi-Status). 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.

8.6.2.1 Example - DELETE

>>Request

```
DELETE  /container/ HTTP/1.1
Host: www.foo.bar
```

>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/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.foo.bar/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.foo.bar/container/resource3 failed because it is locked, and no lock token was submitted with the request. Consequently, the attempt to delete http://www.foo.bar/container/ also failed. Thus the client knows that the attempt to delete http://www.foo.bar/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.

8.7 PUT

8.7.1 PUT for Non-Collection Resources

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

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

8.7.2 PUT for Collections

As defined in the HTTP/1.1 specification [RFC2068], the "PUT method requests that the enclosed entity be stored under the supplied Request-URI." Since submission of an entity representing a collection would implicitly encode creation and deletion of resources, this specification intentionally does not define a transmission format for creating a collection using PUT. Instead, the MKCOL method is defined to create collections.

When the PUT operation creates a new non-collection resource all ancestors MUST already exist. If all ancestors do not exist, the method MUST fail with a 409 (Conflict) status code. For example, if resource /a/b/c/d.html is to be created and /a/b/c/ does not exist, then the request must fail.

8.8 COPY Method

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

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

8.8.1 COPY for HTTP/1.1 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. After a successful COPY invocation, all properties on the source resource MUST be duplicated on the destination resource, subject to modifying headers and XML elements, following the definition for copying properties. 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.

8.8.2 COPY for Properties

The following section defines how properties on a resource are handled during a COPY operation.

Live properties SHOULD be duplicated as identically behaving live properties at the destination resource. If a property cannot be copied live, then its value MUST be duplicated, octet-for-octet, in an identically named, dead property on the destination resource subject to the effects of the propertybehavior XML element.

The propertybehavior XML element can specify that properties are copied on best effort, that all live properties must be successfully copied or the method must fail, or that a specified list of live properties must be successfully copied or the method must fail. The propertybehavior XML element is defined in section 12.12.

8.8.3 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”. DAV compliant servers MUST support the “0” and “infinity” Depth header behaviors.

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.

A COPY of “Depth: 0” only instructs that the collection and its properties but not resources identified by its internal member URIs, 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://fun.com/ and the Destination is http://fun.com/b/ then when http://fun.com/a/c/d is processed it must use a Destination of http://fun.com/b/c/d.

When the COPY method has completed processing it MUST have created a consistent 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).

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.

8.8.4 COPY and the Overwrite Header

If a resource exists at the destination and the Overwrite header is “T” then prior to performing the copy 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.

8.8.5 Status Codes

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.

403 (Forbidden) – The source and destination URIs are the same.

409 (Conflict) – A resource cannot be created at the destination until one or more intermediate collections have been created.

412 (Precondition Failed) - The server was unable to maintain the liveness of the properties listed in the propertybehavior XML element or the Overwrite header is “F” and the state of the destination resource is non-null.

423 (Locked) - The destination resource was locked.

502 (Bad Gateway) - This may occur when the destination is on another server and the destination server refuses to accept the resource.

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

8.8.6 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
```

8.8.7 Example - COPY with No Overwrite

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

>>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
```

8.8.8 Example - COPY of a Collection

>>Request

```
COPY /container/ HTTP/1.1
Host: www.foo.bar
Destination: http://www.foo.bar/othercontainer/
Depth: infinity
Content-Type: text/xml; charset="utf-8"
Content-Length: xxxx

<?xml version="1.0" encoding="utf-8" ?>
<d:propertybehavior xmlns:d="DAV:">
      <d:keepalive>*</d:keepalive>
</d:propertybehavior>
```

>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/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.foo.bar/othercontainer/R2/</d:href>
            <d:status>HTTP/1.1 412 Precondition Failed</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, most likely due to a problem with maintaining the liveness of properties (this is specified by the propertybehavior XML element). 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 given in section 8.8.3.

8.9 MOVE Method

The MOVE operation on a non-collection resource is the logical equivalent of a copy (COPY), followed by consistency maintenance processing, followed by a delete of the source, where all three actions are performed atomically. The consistency maintenance step allows the server to perform updates caused by the move, such as updating all URIs 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 DAV 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.

8.9.1 MOVE for Properties

The behavior of properties on a MOVE, including the effects of the propertybehavior XML element, MUST be the same as specified in section 8.8.2.

8.9.2 MOVE for Collections

A MOVE with "Depth: infinity" instructs that the collection identified by the Request-URI be moved to the URI specified in the Destination header, and all resources identified by its internal member URIs 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 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).

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.

8.9.3 MOVE and the Overwrite Header

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

8.9.4 Status Codes

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

204 (No Content) - The source resource was successfully moved to a pre-existing destination resource.

403 (Forbidden) – The source and destination URIs are the same.

409 (Conflict) – A resource cannot be created at the destination until one or more intermediate collections have been created.

412 (Precondition Failed) - The server was unable to maintain the liveness of the properties listed in the propertybehavior XML element or the Overwrite header is "F" and the state of the destination resource is non-null.

423 (Locked) - The source or the destination resource was locked.

502 (Bad Gateway) - This may occur when the destination is on another server and the destination server refuses to accept the resource.

8.9.5 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 resource had been non-null. 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
```

8.9.6 Example - MOVE of a Collection

>>Request

```
MOVE /container/ HTTP/1.1
Host: www.foo.bar
Destination: http://www.foo.bar/othercontainer/
Overwrite: F
If: (<opaquelocktoken:fe184f2e-6eec-41d0-c765-01adc56e6bb4>)
    (<opaquelocktoken:e454f3f3-acdc-452a-56c7-00a5c91e4b77>)
Content-Type: text/xml; charset="utf-8"
Content-Length: xxxx

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

>>Response

```
HTTP/1.1 207 Multi-Status
Content-Type: text/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.foo.bar/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.foo.bar/othercontainer/C2/. This means that the resource /container/C2/ could not be moved. Because there was an error copying /container/C2/, none of /container/C2's members were copied. However no errors were listed for those members due to the error minimization rules given in section 8.8.3. User agent authentication has previously occurred via a mechanism outside the scope of the HTTP protocol, in an underlying transport layer.

8.10 LOCK Method

The following sections describe the LOCK method, which is used to take out a lock of any access type. 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.

8.10.1 Operation

A LOCK method invocation creates the lock specified by the lockinfo XML element on the Request-URI. Lock method requests SHOULD have a XML request body which contains an owner XML element for this lock request, unless this is a refresh request. 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, an administrator 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. The response MUST contain the value of the lockdiscovery property in a prop XML element.

In order to indicate the lock token associated with a newly created lock, a Lock-Token response header MUST be included in the response for every successful LOCK request for a new lock. Note that the Lock-Token header would not be returned in the response for a successful refresh LOCK request because a new lock was not created.

8.10.2 The Effect of Locks on Properties and Collections

The scope of a lock is the entire state of the resource, including its body and associated properties. As a result, a lock on a resource MUST also lock the resource's properties.

For collections, a lock also affects the ability to add or remove members. The nature of the effect depends upon the type of access control involved.

8.10.3 Locking Replicated Resources

A resource may be made available through more than one URI. However locks apply to resources, not URIs. Therefore a LOCK request on a resource MUST NOT succeed if can not be honored by all the URIs through which the resource is addressable.

8.10.4 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. If the lock cannot be granted to all resources, a 409 (Conflict) status code MUST be returned with a response entity body containing a multistatus XML element describing which resource(s) prevented the lock from being granted. Hence, partial success is not an option. Either the entire hierarchy is locked or no resources are locked.

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

8.10.5 Interaction with other Methods

The interaction of a LOCK with various methods is dependent upon the lock type. However, independent of lock type, a successful DELETE of a resource MUST cause all of its locks to be removed.

8.10.6 Lock Compatibility Table

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

CURRENT LOCK STATE/ LOCK REQUEST	SHARED LOCK	EXCLUSIVE LOCK
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.

8.10.7 Status Codes

200 (OK) - The lock request succeeded and the value of the lockdiscovery property is included in the body.

412 (Precondition Failed) - The included lock token was not enforceable on this resource or the server could not satisfy the request in the lockinfo XML element.

423 (Locked) - The resource is locked, so the method has been rejected.

8.10.8 Example - Simple Lock Request

>>Request

```
LOCK /workspace/webdav/proposal.doc HTTP/1.1
Host: webdav.sb.aol.com
Timeout: Infinite, Second-4100000000
Content-Type: text/xml; charset="utf-8"
Content-Length: xxxx
Authorization: Digest username="ejw",
   realm="ejw@webdav.sb.aol.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
Content-Type: text/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>
              opaquelocktoken:e71d4fae-5dec-22d6-fea5-00a0c91e6be4
                            </D:href>
                     </D:locktoken>
              </D:activelock>
       </D:lockdiscovery>
</D:prop>
```

This example shows the successful creation of an exclusive write lock on resource http://webdav.sb.aol.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.

8.10.9 Example - Refreshing a Write Lock

>>Request

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

>>Response

```
HTTP/1.1 200 OK
Content-Type: text/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>
              opaquelocktoken:e71d4fae-5dec-22d6-fea5-00a0c91e6be4
                            </D:href>
                     </D:locktoken>
              </D:activelock>
       </D:lockdiscovery>
</D:prop>
```

This request would refresh the lock, resetting any time outs. 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.

8.10.10 Example - Multi-Resource Lock Request

>>Request

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

<?xml version="1.0" encoding="utf-8" ?>
<D:multistatus xmlns:D="DAV:">
       <D:response>
              <D:href>http://webdav.sb.aol.com/webdav/secret</D:href>
              <D:status>HTTP/1.1 403 Forbidden</D:status>
       </D:response>
       <D:response>
              <D:href>http://webdav.sb.aol.com/webdav/</D:href>
              <D:propstat>
                     <D:prop><D:lockdiscovery/></D:prop>
                     <D:status>HTTP/1.1 424 Failed Dependency</D:status>
              </D:propstat>
       </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://webdav.sb.aol.com/webdav/secret. Because this resource could not be locked, none of the resources were locked. Note also that the lockdiscovery property for the Request-URI has been included as required. In this example the lockdiscovery property is empty which means that there are no outstanding locks on the resource.

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

8.11 UNLOCK Method

The UNLOCK method removes the lock identified by the lock token in the Lock-Token request header from the Request-URI, and 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.

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

8.11.1 Example - UNLOCK

>>Request

```
UNLOCK /workspace/webdav/info.doc HTTP/1.1
Host: webdav.sb.aol.com
Lock-Token: <opaquelocktoken:a515cfa4-5da4-22e1-f5b5-00a0451e6bf7>
Authorization: Digest username="ejw",
   realm="ejw@webdav.sb.aol.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 “opaquelocktoken:a515cfa4-5da4-22e1-f5b5-00a0451e6bf7” is successfully removed from the resource http://webdav.sb.aol.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

9.1 DAV Header

```
DAV = "DAV" ":" "1" ["," "2"] ["," 1#extend]
```

This header indicates that the resource supports the DAV schema and protocol as specified. All DAV compliant resources MUST return the DAV header on all OPTIONS responses.

The value is a list of all compliance classes that the resource supports. Note that above a comma has already been added to the 2. This is because a resource can not be level 2 compliant unless it is also level 1 compliant. Please refer to section 15 for more details. In general, however, support for one compliance class does not entail support for any other.

9.2 Depth Header

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

The Depth 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" ":" absoluteURI
```

The Destination 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 absoluteURI production is defined in [RFC2396].

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 "]")) ")"
State-token = Coded-URL
Coded-URL = "<" absoluteURI ">"
```

The If header is intended to have similar functionality to the If-Match header defined in section 14.25 of [RFC2068]. 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.

All DAV compliant resources MUST honor the If header.

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.

Note that the absoluteURI production is defined in [RFC2396].

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.

If a method, due to the presence of a Depth or Destination header, is applied to multiple resources then the No-tag-list production MUST be applied to each resource the method is applied to.

9.4.1.1 Example - No-tag-list If Header

```
If: (<locktoken:a-write-lock-token> ["I am an ETag"]) (["I am another
ETag"])
```

The previous header would require that any resources within the scope of the method must either 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 <locktoken:a-write-lock-token> [“I am an ETag”]) (and [“I am another ETag”])).

9.4.2 Tagged-list Production

The tagged-list production scopes a list production. 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.

When the If header is applied to a particular resource, the Tagged-list productions MUST be searched to determine if any of the listed resources match the operand resource(s) for the current method. If none of the resource productions match the current resource then the header MUST be ignored. If one of the resource productions does match the name of the resource under consideration then the list productions following the resource production MUST be applied to the resource in the manner specified in the previous section.

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

9.4.2.1 Example - Tagged List If header

```
COPY /resource1 HTTP/1.1
Host: www.foo.bar
Destination: http://www.foo.bar/resource2
If: <http://www.foo.bar/resource1> (<locktoken:a-write-lock-token>
[W/"A weak ETag"]) (["strong ETag"])
<http://www.bar.bar/random>(["another strong ETag"])
```

In this example http://www.foo.bar/resource1 is being copied to http://www.foo.bar/resource2. When the method is first applied to http://www.foo.bar/resource1, resource1 must be in the state specified by "(<locktoken:a-write-lock-token> [W/"A weak ETag"]) (["strong ETag"])", that is, it either must be locked with a lock token of "locktoken:a-write-lock-token" 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.bar.bar/random never has the method applied to it (the only other resource listed in the If header) and http://www.foo.bar/resource2 is not listed in the If header.

9.4.3 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 <locktoken:write1> <locktoken:write2>)
```

When submitted with a request, this If header requires that all operand resources must not be locked with locktoken:write1 and must be locked with locktoken:write2.

9.4.4 Matching Function

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

Matching entity tag: Where the entity tag matches an entity tag associated with that resource.

Matching state token: Where there is an exact match between the state token in the If header and any state token on the resource.

9.4.5 If Header and Non-DAV Compliant Proxies

Non-DAV compliant 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 header specifies whether the server should overwrite the state of a non-null destination resource 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 resource is non-null. 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.

All DAV compliant resources MUST support the Overwrite header.

9.7 Status-URI Response Header

The Status-URI response header may be used with the 102 (Processing) status code to inform the client as to the status of a method.

```
Status-URI = "Status-URI" ":" *(Status-Code Coded-URL) ; Status-Code is
defined in 6.1.1 of [RFC2068]
```

The URIs listed in the header are source resources which have been affected by the outstanding method. The status code indicates the resolution of the method on the identified resource. So, for example, if a MOVE method on a collection is outstanding and a 102 (Processing) response with a Status-URI response header is returned, the included URIs will indicate resources that have had move attempted on them and what the result was.

9.8 Timeout Request Header

```
TimeOut = "Timeout" ":" 1#TimeType
TimeType = ("Second-" DAVTimeOutVal | "Infinite" | Other)
DAVTimeOutVal = 1*digit
Other = "Extend" field-value   ; See section 4.2 of [RFC2068]
```

Clients may include Timeout 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.

A Timeout request header MUST contain at least one TimeType and may contain multiple TimeType entries. The purpose of listing multiple TimeType entries is to indicate multiple different values and value types that are acceptable to the client. The client lists the TimeType entries in order of preference.

Timeout response values MUST use a Second value, Infinite, or a TimeType the client has indicated familiarity with. The server may assume a client is familiar with any TimeType submitted in a Timeout header.

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 SHOULD be restarted any time an owner of the lock sends a method to any member of the lock, including unsupported methods, or methods which are unsuccessful. However the lock MUST be refreshed if a refresh LOCK method is successfully received.

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.

10 Status Code Extensions to HTTP/1.1

The following status codes are added to those defined in HTTP/1.1 [RFC2068].

10.1 102 Processing

The 102 (Processing) status code is an interim response used to inform the client that the server has accepted the complete request, but has not yet completed it. This status code SHOULD only be sent when the server has a reasonable expectation that the request will take significant time to complete. As guidance, if a method is taking longer than 20 seconds (a reasonable, but arbitrary value) to process the server SHOULD return a 102 (Processing) response. The server MUST send a final response after the request has been completed.

Methods can potentially take a long period of time to process, especially methods that support the Depth header. In such cases the client may time-out the connection while waiting for a response. To prevent this the server may return a 102 (Processing) status code to indicate to the client that the server is still processing the method.

10.2 207 Multi-Status

The 207 (Multi-Status) status code provides status for multiple independent operations (see section 11 for more information).

10.3 422 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.

10.4 423 Locked

The 423 (Locked) status code means the source or destination resource of a method is locked.

10.5 424 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).

10.6 507 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 Multi-Status Response

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.

12 XML Element Definitions

In the section below, the final line of each section gives the element type declaration using the format defined in [REC-XML]. The "Value" field, where present, specifies further restrictions on the allowable contents of the XML element using BNF (i.e., to further restrict the values of a PCDATA element).

12.1 activelock XML Element

Name: activelock
Namespace: DAV:
Purpose: Describes a lock on a resource.

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

12.1.1 depth XML Element

Name: depth
Namespace: DAV:
Purpose: The value of the Depth header.
Value: `"0" | "1" | "infinity"`

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

12.1.2 locktoken XML Element

Name: locktoken
Namespace: DAV:
Purpose: The lock token associated with a lock.
Description: The href contains one or more opaque lock token URIs which all refer to the same lock (i.e., the OpaqueLockToken-URI production in section 6.4).

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

12.1.3 timeout XML Element

Name: timeout
Namespace: DAV:
Purpose: The timeout associated with a lock
Value: `TimeType ;Defined in section 9.8`

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

12.2 collection XML Element

Name: collection
Namespace: DAV:
Purpose: Identifies the associated resource as a collection. The resourcetype property of a collection resource MUST have this value.

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

12.3 href XML Element

Name: href
Namespace: DAV:
Purpose: Identifies the content of the element as a URI.
Value: `URI ; See section 3.2.1 of [RFC2068]`

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

12.4 link XML Element

Name: link
Namespace: DAV:
Purpose: Identifies the property as a link and contains the source and destination of that link.
Description: The link XML element is used to provide the sources and destinations of a link. The name of the property containing the link XML element provides the type of the link. Link is a multi-valued element, so multiple links may be used together to indicate multiple links with the same type. The values in the href XML elements inside the src and dst XML elements of the link XML element MUST NOT be rejected if they point to resources which do not exist.

```
<!ELEMENT link (src+, dst+) >
```

12.4.1 dst XML Element

Name: dst
Namespace: DAV:
Purpose: Indicates the destination of a link
Value: `URI`

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

12.4.2 src XML Element

Name: src
Namespace: DAV:
Purpose: Indicates the source of a link.
Value: URI

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

12.5 lockentry XML Element

Name: lockentry
Namespace: DAV:
Purpose: Defines the types of locks that can be used with the resource.

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

12.6 lockinfo XML Element

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

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

12.7 lockscope XML Element

Name: lockscope
Namespace: DAV:
Purpose: Specifies whether a lock is an exclusive lock, or a shared lock.

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

12.7.1 exclusive XML Element

Name: exclusive
Namespace: DAV:
Purpose: Specifies an exclusive lock

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

12.7.2 shared XML Element

Name: shared
Namespace: DAV:
Purpose: Specifies a shared lock

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

12.8 locktype XML Element

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

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

12.8.1 write XML Element

Name: write
Namespace: DAV:
Purpose: Specifies a write lock.

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

12.9 multistatus XML Element

Name: multistatus
Namespace: DAV:
Purpose: Contains multiple response messages.
Description: The responsedescription at the top level is used to provide a general message describing the overarching nature of the response. If this value is available an application may use it instead of presenting the individual response descriptions contained within the responses.

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

12.9.1 response XML Element

Name: response
Namespace: DAV:
Purpose: Holds a single response describing the effect of a method on resource and/or its properties.
Description: A particular href 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.

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

12.9.1.1 propstat XML Element

Name: propstat
Namespace: DAV:
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.

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

12.9.1.2 status XML Element

Name: status
Namespace: DAV:
Purpose: Holds a single HTTP status-line
Value: `status-line ;status-line defined in [RFC2068]`

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

12.9.2 responsedescription XML Element

Name: responsedescription
Namespace: DAV:
Purpose: Contains a message that can be displayed to the user explaining the nature of the response.
Description: This XML element provides information suitable to be presented to a user.

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

12.10 owner XML Element

Name: owner
Namespace: DAV:
Purpose: Provides information about the principal taking out a lock.
Description: The owner XML element 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.

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

12.11 prop XML element

Name: prop
Namespace: DAV:
Purpose: Contains properties related to a resource.
Description: The prop XML element is 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.

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

12.12 propertybehavior XML element

Name: propertybehavior
Namespace: DAV:
Purpose: Specifies how properties are handled during a COPY or MOVE.
Description: The propertybehavior XML element specifies how properties are handled during a COPY or MOVE. If this XML element is not included in the request body then the server is expected to act as defined by the default property handling behavior of the associated method. All WebDAV compliant resources MUST support the propertybehavior XML element.

```
<!ELEMENT propertybehavior (omit | keepalive) >
```

12.12.1 keepalive XML element

Name: keepalive
Namespace: DAV:
Purpose: Specifies requirements for the copying/moving of live properties.
Description: If a list of URIs is included as the value of keepalive then the named properties MUST be “live” after they are copied (moved) to the destination resource of a COPY (or MOVE). If the value “*” is given for the keepalive XML element, this designates that all live properties on the source resource MUST be live on the destination. If the requirements specified by the keepalive element can not be honored then the method MUST fail with a 412 (Precondition Failed). All DAV compliant resources MUST support the keepalive XML element for use with the COPY and MOVE methods.
Value: `"*" ; #PCDATA value can only be "*"`

```
<!ELEMENT keepalive (#PCDATA | href+) >
```

12.12.2 omit XML element

Name: omit
Namespace: DAV:
Purpose: The omit XML element instructs the server that it should use best effort to copy properties but a failure to copy a property MUST NOT cause the method to fail.
Description: The default behavior for a COPY or MOVE is to copy/move all properties or fail the method. In certain circumstances, such as when a server copies a resource over another protocol such as FTP, it may not be possible to copy/move the properties associated with the resource. Thus any attempt to copy/move over FTP would always have to fail because properties could not be moved over, even as dead properties. All DAV compliant resources MUST support the omit XML element on COPY/MOVE methods.

```
<!ELEMENT omit EMPTY >
```

12.13 propertyupdate XML element

Name: propertyupdate
Namespace: DAV:
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.

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

12.13.1 remove XML element

Name: remove
Namespace: DAV:
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.

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

12.13.2 set XML element

Name: set
Namespace: DAV:
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 in the property's value (in the “xml:lang” attribute, if present) MUST be persistently stored along with the property, and MUST be subsequently retrievable using PROPFIND.

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

12.14 propfind XML Element

Name: propfind
Namespace: DAV:
Purpose: Specifies the properties to be returned from a PROPFIND method. Two special elements are specified for use with propfind, allprop and propname. If prop is used inside propfind it MUST only contain property names, not values.

```
<!ELEMENT propfind (allprop | propname | prop) >
```

12.14.1 allprop XML Element

Name: allprop
Namespace: DAV:
Purpose: The allprop XML element specifies that all property names and values on the resource are to be returned.

```
<!ELEMENT allprop EMPTY >
```

12.14.2 propname XML Element

Name: propname
Namespace: DAV:
Purpose: The propname XML element specifies that only a list of property names on the resource is to be returned.

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

13 DAV Properties

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

13.1 creationdate Property

Name:	creationdate
Namespace:	DAV:
Purpose:	Records the time and date the resource was created.
Value:	`date-time` ; See Appendix 2
Description:	The creationdate property should be defined on all DAV compliant resources. If present, it contains a timestamp of the moment when the resource was created (i.e., the moment it had non-null state).

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

13.2 displayname Property

Name:	displayname
Namespace:	DAV:
Purpose:	Provides a name for the resource that is suitable for presentation to a user.
Description:	The 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.

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

13.3 getcontentlanguage Property

Name:	getcontentlanguage
Namespace:	DAV:
Purpose:	Contains the Content-Language header returned by a GET without accept headers
Description:	The getcontentlanguage property MUST be defined on any DAV compliant resource that returns the Content-Language header on a GET.
Value:	`language-tag` ;language-tag is defined in section 14.13 of [RFC2068]

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

13.4 getcontentlength Property

Name:	getcontentlength
Namespace:	DAV:
Purpose:	Contains the Content-Length header returned by a GET without accept headers.
Description:	The getcontentlength property MUST be defined on any DAV compliant resource that returns the Content-Length header in response to a GET.
Value:	`content-length` ; see section 14.14 of [RFC2068]

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

13.5 getcontenttype Property

Name:	getcontenttype
Namespace:	DAV:
Purpose:	Contains the Content-Type header returned by a GET without accept headers.
Description:	This getcontenttype property MUST be defined on any DAV compliant resource that returns the Content-Type header in response to a GET.
Value:	`media-type` ; defined in section 3.7 of [RFC2068]

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

13.6 getetag Property

Name:	getetag
Namespace:	DAV:
Purpose:	Contains the ETag header returned by a GET without accept headers.
Description:	The getetag property MUST be defined on any DAV compliant resource that returns the Etag header.
Value:	`entity-tag` ; defined in section 3.11 of [RFC2068]

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

13.7 getlastmodified Property

Name:	getlastmodified
Namespace:	DAV:
Purpose:	Contains the Last-Modified header returned by a GET method without accept headers.
Description:	Note that the last-modified date on a resource may reflect changes in any part of the state of the resource, not necessarily just a change to the response to the GET method. For example, a change in a property may cause the last-modified date to change. The getlastmodified property MUST be defined on any DAV compliant resource that returns the Last-Modified header in response to a GET.
Value:	`HTTP-date` ; defined in section 3.3.1 of [RFC2068]

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

13.8 lockdiscovery Property

Name:	lockdiscovery
Namespace:	DAV:
Purpose:	Describes the active locks on a resource
Description:	The lockdiscovery property 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. The server is free to withhold any or all of this information if the requesting principal does not have sufficient access rights to see the requested data.

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

13.8.1 Example - Retrieving the lockdiscovery Property

>>Request

```
PROPFIND /container/ HTTP/1.1
Host: www.foo.bar
Content-Length: xxxx
Content-Type: text/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: text/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.foo.bar/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>
            opaquelocktoken:f81de2ad-7f3d-a1b2-4f3c-00a0c91a9d76
                                     </D:href>
                               </D:locktoken>
                           </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.

13.9 resourcetype Property

Name:	resourcetype
Namespace:	DAV:
Purpose:	Specifies the nature of the resource.
Description:	The resourcetype property MUST be defined on all DAV compliant resources. The default value is empty.

```
<!ELEMENT resourcetype ANY >
```

13.10 source Property

Name: source
Namespace: DAV:
Purpose: The destination of the source link identifies the resource that contains the unprocessed source of the link's source.
Description: The source of the link (src) is typically the URI of the output resource on which the link is defined, and there is typically only one destination (dst) of the link, which is the URI where the unprocessed source of the resource may be accessed. When more than one link destination exists, this specification asserts no policy on ordering.

```
<!ELEMENT source (link)* >
```

13.10.1 Example - A source Property

```
<?xml version="1.0" encoding="utf-8" ?>
<D:prop xmlns:D="DAV:" xmlns:F="http://www.foocorp.com/Project/">
       <D:source>
              <D:link>
                     <F:projfiles>Source</F:projfiles>
                     <D:src>http://foo.bar/program</D:src>
                     <D:dst>http://foo.bar/src/main.c</D:dst>
              </D:link>
              <D:link>
                     <F:projfiles>Library</F:projfiles>
                     <D:src>http://foo.bar/program</D:src>
                     <D:dst>http://foo.bar/src/main.lib</D:dst>
              </D:link>
              <D:link>
                     <F:projfiles>Makefile</F:projfiles>
                     <D:src>http://foo.bar/program</D:src>
                     <D:dst>http://foo.bar/src/makefile</D:dst>
              </D:link>
       </D:source>
</D:prop>
```

In this example the resource http://foo.bar/program has a source property that contains three links. Each link contains three elements, two of which, src and dst, are part of the DAV schema defined in this document, and one which is defined by the schema http://www.foocorp.com/project/ (Source, Library, and Makefile). A client which only implements the elements in the DAV spec will not understand the foocorp elements and will ignore them, thus seeing the expected source and destination links. An enhanced client may know about the foocorp elements and be able to present the user with additional information about the links. This example demonstrates the power of XML markup, allowing element values to be enhanced without breaking older clients.

13.11 supportedlock Property

Name: supportedlock
Namespace: DAV:
Purpose: To provide a listing of the lock capabilities supported by the resource.
Description: The supportedlock property of a resource 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.

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

13.11.1 Example - Retrieving the supportedlock Property

>>Request

```
PROPFIND  /container/ HTTP/1.1
Host: www.foo.bar
Content-Length: xxxx
Content-Type: text/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: text/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.foo.bar/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>
```

14 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 DAV client MUST accept XML responses using either text/xml or application/xml.

15 DAV Compliance Classes

A DAV compliant resource can choose from two 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.

Since this document describes extensions to the HTTP/1.1 protocol, minimally all DAV compliant resources, clients, and proxies MUST be compliant with [RFC2068].

Compliance classes are not necessarily sequential. A resource that is class 2 compliant must also be class 1 compliant; but if additional compliance classes are defined later, a resource that is class 1, 2, and 4 compliant might not be class 3 compliant. Also note that identifiers other than numbers may be used as compliance class identifiers.

15.1 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.

15.2 Class 2

A class 2 compliant resource MUST meet all class 1 requirements and support the LOCK method, the supportedlock property, the 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.

16 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 [UTF-8] encoding 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 [RFC2376], as well as the XML "encoding" attribute, which together 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. XML uses either IANA registered language tags (see [RFC1766]) or ISO 639 language tags [ISO-639] in the "xml:lang" attribute of an XML element to identify the language of its content and attributes.

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" [RFC2376] 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 three categories: names of protocol elements such as methods and headers, names of XML elements, and names of properties. 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 in fact simply long token identifiers, they do not need to support encoding in multiple character sets. Similarly, though the names of XML elements used in this specification are English names encoded in UTF-8, these names are not visible to the user, and hence do not need to support multiple character set encodings.

The name of a property defined on a resource is a URI. Although some applications (e.g., a generic property viewer) will display property URIs 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 URI 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 URI 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.

17 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 [RFC2068]) and XML (discussed in [RFC2376]) 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.

17.1 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 [RFC2069]. 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.

17.2 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.

17.3 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.

17.4 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 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 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.

17.5 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.

17.6 Reduction of Security due to Source Link

HTTP/1.1 warns against providing read access to script code because it may contain sensitive information. Yet WebDAV, via its source link facility, can potentially provide a URI for script resources so they may be authored. For HTTP/1.1, a server could reasonably prevent access to source resources due to the predominance of read-only access. WebDAV, with its emphasis on authoring, encourages read and write access to source resources, and provides the source link facility to identify the source. This reduces the security benefits of eliminating access to source resources. Users and administrators of WebDAV servers should be very cautious when allowing remote authoring of scripts, limiting read and write access to the source resources to authorized principals.

17.7 Implications of XML External Entities

XML supports a facility known as “external entities”, defined in section 4.2.2 of [REC-XML], which instruct an XML processor to retrieve and perform an inline include of XML located at a particular URI. 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 [REC-XML]. However, [REC-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 [RFC2376]. Therefore, implementers must be aware that external XML entities should be treated as untrustworthy.

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.

17.8 Risks Connected with Lock Tokens

This specification, in section 6.4, requires the use of Universal Unique Identifiers (UUIDs) for lock tokens, in order to guarantee their uniqueness across space and time. UUIDs, as defined in [ISO-11578], contain a “node” field which “consists of the IEEE address, usually the host address. For systems with multiple IEEE 802 nodes, any available node address can be used.” Since a WebDAV server will issue many locks over its lifetime, the implication is that it will also be publicly exposing its IEEE 802 address.

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.
Section 6.4.1 of this specification details an alternate mechanism for generating the “node” field of a UUID without using an IEEE 802 address, which alleviates the risks associated with exposure of IEEE 802 addresses by using an alternate source of uniqueness.

18 IANA Considerations

This document defines two namespaces, the namespace of property names, and the namespace of WebDAV-specific XML elements used within property values.

URIs are used for both names, for several reasons. Assignment of a URI does not require a request to a central naming authority, and hence allow WebDAV property names and XML elements to be quickly defined by any WebDAV user or application. URIs also provide a unique address space, ensuring that the distributed users of WebDAV will not have collisions among the property names and XML elements they create.

This specification defines a distinguished set of property names and XML elements that are understood by all WebDAV applications. The property names and XML elements in this specification are all derived from the base URI DAV: by adding a suffix to this URI, for example, DAV:creationdate for the “creationdate” property.

This specification also defines a URI scheme for the encoding of lock tokens, the opaquelocktoken URI scheme described in section 6.4.

To ensure correct interoperation based on this specification, IANA must reserve the URI namespaces starting with “DAV:” and with “opaquelocktoken:” for use by this specification, its revisions, and related WebDAV specifications.

19 Intellectual Property

The following notice is copied from RFC 2026 [RFC2026], section 10.4, and describes the position of the IETF concerning intellectual property claims made against this document.

The IETF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use other technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on the IETF's procedures with respect to rights in standards-track and standards-related documentation can be found in BCP-11. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification can be obtained from the IETF Secretariat.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the IETF Executive Director.

20 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.

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.

21 References

21.1 Normative References

[RFC1766] H. T. Alvestrand, “Tags for the Identification of Languages.” RFC 1766. Uninett. March, 1995.

[RFC2277] H. T. Alvestrand, “IETF Policy on Character Sets and Languages.” RFC 2277, BCP 18. Uninett. January, 1998.

[RFC2119] S. Bradner, “Key words for use in RFCs to Indicate Requirement Levels.” RFC 2119, BCP 14. Harvard University. March, 1997.

[RFC2396] T. Berners-Lee, R. Fielding, L. Masinter, “Uniform Resource Identifiers (URI): Generic Syntax.” RFC 2396. MIT/LCS, U.C. Irvine, Xerox. August, 1998.

[REC-XML]T. Bray, J. Paoli, C. M. Sperberg-McQueen, “Extensible Markup Language (XML).” World Wide Web Consortium Recommendation REC-xml-19980210. http://www.w3.org/TR/1998/REC-xml-19980210.

[REC-XML-NAMES] T. Bray, D. Hollander, A. Layman, “Name Spaces in XML” World Wide Web Consortium Recommendation REC-xml-names. http://www.w3.org/TR/REC-xml-names-19990114/

[RFC2069] J. Franks, P. Hallam-Baker, J. Hostetler, P. Leach, A. Luotonen, E. Sink, and L. Stewart. “An Extension to HTTP : Digest Access Authentication” RFC 2069. Northwestern University, CERN, Spyglass Inc., Microsoft Corp., Netscape Communications Corp., Spyglass Inc., Open Market Inc. January 1997.

[RFC2068] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1.” RFC 2068. U.C. Irvine, DEC, MIT/LCS. January, 1997.

[ISO-639] ISO (International Organization for Standardization). ISO 639:1988. “Code for the representation of names of languages.”

[ISO-8601] ISO (International Organization for Standardization). ISO 8601:1988. “Data elements and interchange formats - Information interchange - Representation of dates and times.”

[ISO-11578] ISO (International Organization for Standardization). ISO/IEC 11578:1996. “Information technology - Open Systems Interconnection - Remote Procedure Call (RPC)”

[RFC2141] R. Moats, “URN Syntax.” RFC 2141. AT&T. May, 1997.

[UTF-8] F. Yergeau, “UTF-8, a transformation format of Unicode and ISO 10646.” RFC 2279. Alis Technologies. January, 1998.

21.2 Informational References

[RFC2026] S. Bradner, “The Internet Standards Process - Revision 3.” RFC 2026, BCP 9. Harvard University. October, 1996.

[RFC1807] R. Lasher, D. Cohen, “A Format for Bibliographic Records,” RFC 1807. Stanford, Myricom. June, 1995.

[WF] C. Lagoze, “The Warwick Framework: A Container Architecture for Diverse Sets of Metadata”, D-Lib Magazine, July/August 1996. http://www.dlib.org/dlib/july96/lagoze/07lagoze.html

[USMARC] Network Development and MARC Standards, Office, ed. 1994. “USMARC Format for Bibliographic Data”, 1994. Washington, DC: Cataloging Distribution Service, Library of Congress.

[REC-PICS] J. Miller, T. Krauskopf, P. Resnick, W. Treese, “PICS Label Distribution Label Syntax and Communication Protocols” Version 1.1, World Wide Web Consortium Recommendation REC-PICS-labels-961031. http://www.w3.org/pub/WWW/TR/REC-PICS-labels-961031.html.

[RFC2291] J. A. Slein, F. Vitali, E. J. Whitehead, Jr., D. Durand, “Requirements for Distributed Authoring and Versioning Protocol for the World Wide Web.” RFC 2291. Xerox, Univ. of Bologna, U.C. Irvine, Boston Univ. February, 1998.

[RFC2413] S. Weibel, J. Kunze, C. Lagoze, M. Wolf, “Dublin Core Metadata for Resource Discovery.” RFC 2413. OCLC, UCSF, Cornell, Reuters. September, 1998.

[RFC2376] E. Whitehead, M. Murata, “XML Media Types.” RFC 2376. U.C. Irvine, Fuji Xerox Info. Systems. July 1998.

22 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

23 Appendices

23.1 Appendix 1 - WebDAV Document Type Definition

This section provides a document type definition, following the rules in [REC-XML], for the XML elements used in the protocol stream and in the values of properties. It collects the element definitions given in sections 12 and 13.

```
<!DOCTYPE webdav-1.0 [

<!--============ XML Elements from Section 12 =================-->

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

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

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

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

<!ELEMENT depth (#PCDATA) >

<!ELEMENT owner ANY >

<!ELEMENT timeout (#PCDATA) >

<!ELEMENT locktoken (href+) >

<!ELEMENT href (#PCDATA) >

<!ELEMENT link (src+, dst+) >
<!ELEMENT dst (#PCDATA) >
<!ELEMENT src (#PCDATA) >

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

<!ELEMENT response (href, ((href*, status)|(propstat+)),
responsedescription?) >
<!ELEMENT status (#PCDATA) >
<!ELEMENT propstat (prop, status, responsedescription?) >
<!ELEMENT responsedescription (#PCDATA) >

<!ELEMENT prop ANY >

<!ELEMENT propertybehavior (omit | keepalive) >
<!ELEMENT omit EMPTY >
<!ELEMENT keepalive (#PCDATA | href+) >

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

<!ELEMENT propfind (allprop | propname | prop) >
<!ELEMENT allprop EMPTY >
<!ELEMENT propname EMPTY >

<!ELEMENT collection EMPTY >

<!--============ Property Elements from Section 13 ===============-->
<!ELEMENT creationdate (#PCDATA) >
<!ELEMENT displayname (#PCDATA) >
```

```
<!ELEMENT getcontentlanguage (#PCDATA) >
<!ELEMENT getcontentlength (#PCDATA) >
<!ELEMENT getcontenttype (#PCDATA) >
<!ELEMENT getetag (#PCDATA) >
<!ELEMENT getlastmodified (#PCDATA) >
<!ELEMENT lockdiscovery (activelock)* >
<!ELEMENT resourcetype ANY >
<!ELEMENT source (link)* >
<!ELEMENT supportedlock (lockentry)* >
]>
```

23.2 Appendix 2 - ISO 8601 Date and Time Profile

The creationdate property specifies the use of the ISO 8601 date format [ISO-8601]. This section defines a profile of the ISO 8601 date format for use with this specification. This profile is quoted from an Internet-Draft by Chris Newman, and is mentioned here to properly attribute his work.

```
date-time       = full-date "T" full-time

full-date       = date-fullyear "-" date-month "-" date-mday
full-time       = partial-time time-offset

date-fullyear   = 4DIGIT
date-month      = 2DIGIT  ; 01-12
date-mday       = 2DIGIT  ; 01-28, 01-29, 01-30, 01-31 based on month/year
time-hour       = 2DIGIT  ; 00-23
time-minute     = 2DIGIT  ; 00-59
time-second     = 2DIGIT  ; 00-59, 00-60 based on leap second rules
time-secfrac    = "." 1*DIGIT
time-numoffset  = ("+" / "-") time-hour ":" time-minute
time-offset     = "Z" / time-numoffset

partial-time    = time-hour ":" time-minute ":" time-second
                  [time-secfrac]
```

Numeric offsets are calculated as local time minus UTC (Coordinated Universal Time). So the equivalent time in UTC can be determined by subtracting the offset from the local time. For example, 18:50:00-04:00 is the same time as 22:58:00Z.

If the time in UTC is known, but the offset to local time is unknown, this can be represented with an offset of “-00:00”. This differs from an offset of “Z” which implies that UTC is the preferred reference point for the specified time.

23.3 Appendix 3 - Notes on Processing XML Elements

23.3.1 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.

It is a violation of the XML specification to use the <A></A> form if the associated DTD declares the element to be EMPTY (e.g., <!ELEMENT A EMPTY>). If such a statement is included, then the empty element format, <A/> must be used. If the element is not declared to be EMPTY, then either form <A></A> or <A/> may be used for empty elements.

23.3.2 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.

23.3.2.1 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.

23.3.2.2 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.foo.bar/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.foo.bar/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 14, 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.foo.bar/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.

23.4 Appendix 4 -- XML Namespaces for WebDAV

23.4.1 Introduction

All DAV compliant systems MUST support the XML namespace extension as specified in [REC-XML-NAMES].

23.4.2 Meaning of Qualified Names

[Note to the reader: This section does not appear in [REC-XML-NAMES], but is necessary to avoid ambiguity for WebDAV XML processors.]

WebDAV compliant XML processors MUST interpret a qualified name as a URI constructed by appending the LocalPart to the namespace name URI.

Example

```
<del:glider xmlns:del="http://www.del.jensen.org/">
      <del:glidername>
            Johnny Updraft
      </del:glidername>
      <del:glideraccidents/>
</del:glider>
```

In this example, the qualified element name “del:glider” is interpreted as the URL “http://www.del.jensen.org/glider”.

```
<bar:glider xmlns:del="http://www.del.jensen.org/">
      <bar:glidername>
            Johnny Updraft
      </bar:glidername>
      <bar:glideraccidents/>
</bar:glider>
```

Even though this example is syntactically different from the previous example, it is semantically identical. Each instance of the namespace name “bar” is replaced with “http://www.del.jensen.org/” and then appended to the local name for each element tag. The resulting tag names in this example are exactly the same as for the previous example.

```
<foo:r xmlns:foo="http://www.del.jensen.org/glide">
       <foo:rname>
              Johnny Updraft
       </foo:rname>
       <foo:raccidents/>
</foo:r>
```

This example is semantically identical to the two previous ones. Each instance of the namespace name “foo” is replaced with “http://www.del.jensen.org/glide” which is then appended to the local name for each element tag, the resulting tag names are identical to those in the previous examples.

24 Full Copyright Statement