The WebDAV Property Design

E. James Whitehead, Jr.[*], Yaron Y. Goland

ejw@cs.ucsc.edu, yaron@goland.org
[*] Dept. of Computer Science
Univ. of California, Santa Cruz
Santa Cruz, California 95064

Abstract

This paper provides a detailed description of the general design space for metadata storage capabilities. The design space considers issues of metadata identification, typing and representation, dynamic behavior, predefined and user defined metadata, schema discovery/update, operations, API packaging/marshalling, searching, and versioning. The design space is used to structure a retrospective analysis of the three major alternative metadata designs considered during the design of the WebDAV Distributed Authoring Protocol. Deployment experience with WebDAV properties is also discussed, with the most successful use occurring in custom client/server pairs and in protocol extensions.

Keywords: metadata storage and retrieval, metadata design, WebDAV, Web authoring, document management

Introduction

The World Wide Web's dramatic success at providing large amounts of read-only information and form-based services raises the question of how Web pages can be authored remotely, making it as easy to write content to the Web as it is to read it. Rising to this challenge, in 1997 the Internet Engineering Task Force (IETF) chartered the Web Distributed Authoring and Versioning (WebDAV) working group with the goal of making remote collaborative Web authoring tools broadly interoperable. Over a period of two and a half years, the WebDAV working group explored a series of protocol design alternatives, beginning in the summer of 1996 and culminating in the publication of the WebDAV Distributed Authoring Protocol in February 1999 [18].

The WebDAV Distributed Authoring Protocol (hereafter the "WebDAV protocol") is a set of extensions to the Hypertext Transfer Protocol (HTTP) [16], providing three main sets of capabilities:

- *Metadata management:* the ability to read, write, and modify static and dynamic properties (metadata items) on Web resources.
- *Namespace management:* the ability to organize resources into collection hierarchies. Specifically, the ability to create, list, and delete collections, and to copy and move individual resources, as well as trees of collections and their contained resources.
- *Overwrite prevention:* a write locking facility intended to prevent overwrite conflicts. Locks may be scoped to individual resources, or to entire collection hierarchies. Shared write locks are also supported, though infrequently used.

Today, the WebDAV protocol is mature and supported in a broad range of applications and servers. Authoring applications such as Word, FrameMaker, InDesign, OpenOffice, Excel, PowerPoint, Photoshop, and Illustrator all support WebDAV-based collaborative authoring. Web site authoring tools such as Dreamweaver and GoLive also support WebDAV, as do the XML authoring tools XMLSpy, XMLMind, and XMetal. Multiple clients map a WebDAV repository to a disk drive, including Apple OS X (webdavfs), Microsoft Web Folders, WebDrive, Xythos WebFile Client, and DAVfs for Linux. On the server side, WebDAV is supported in Apache (mod_dav and Slide), Microsoft IIS 5/6, Exchange, SharePoint, Oracle 9i (AS/FS), Xythos WebFile Server, Lotus Domino 6, WebStar V, FileNet Panagon, Tamino, Intraspect, CoreMedia, and many others.

To enhance the abilities of the core WebDAV protocol, several follow-on protocols have built upon it. DeltaV adds versioning and configuration management features to WebDAV, making it possible to support remote collaboration on large software projects or collections of documents [8]. DASL (DAV Searching and Locating) unlocks the potential of WebDAV metadata by supporting queries over property values [27]. An access control protocol permits the setting of access control lists on resources to control who can perform specific operations [9]. These protocols have either been recently approved (DeltaV) or are still being developed (DASL, ACL), and hence not yet widely adopted.

In previous work we summarized the WebDAV protocol, and briefly described some design rationale [39]. In this paper, we focus on the metadata features of the WebDAV protocol, with a goal of describing the range of considered design alternatives. The WebDAV protocol is a good artifact to use for exploring these design tradeoffs, since the IETF requirements of openness and transparency led to a design process with written documentation on multiple approaches (in the form of a series of protocol specification drafts), and

years of archived mailing list design discussions. In this paper we distill these drafts and design discussions into a detailed description of design spaces and alternatives that, had they existed beforehand, would have saved much time in the development of WebDAV. Furthermore, we do so in a sufficiently general way that the results are relevant to designers of any other system that uses similar capabilities.

In the following section we detail the metadata capabilities of the WebDAV protocol to provide context and background for the remainder of the paper. Next, we describe real-world experience from the deployment of WebDAV property capability. Following, we describe a general design space for metadata storage that applies to all systems with metadata support. We next describe the main metadata design alternatives considered by the WebDAV protocol, using them as example instances of points in the metadata design space.

Metadata in WebDAV

In the course of their personal and work activities, people use and create a large variety of intellectual works, including musical performances, movies, magazine and journal articles, books, business forms, photographs, posters, and so on. Each of these works has one or more representations of its content, be it a bound paper book, a PDF version of a business form, or an MP3 encoding of a song. Using the terminology of the Web, these representations are known as *resources*. For a variety of reasons, it is useful to be able to associate information with a work's representation (with a resource), without embedding the information in the representation itself.

Several goals motivated the metadata support capabilities of the WebDAV protocol. The ability to associate metadata with resources is valuable for a wide range of document management activities, such as recording workflow state information, and tailoring the system for specific document processing applications. The ability to record bibliographic metadata, such as Dublin Core metadata items [40], and content rating metadata, specifically the Platform for Internet Content Selection (PICS) [22], were explicitly mentioned in the WebDAV protocol requirements document [33]. Version control systems invariably associate metadata with revisions to represent revision identifiers, comments, predecessor and successor relationships, and revision labels. Providing access to typical filesystem metadata, such as the creation date, size, and last modified date is also desirable.

Consider a document management example involving metadata associated with digitally scanned paper insurance claim forms. To process each claim, it is necessary to extract the name, address, and account number of the claimant from the form. Each claim is automatically assigned a tracking number. As the claim is reviewed and approved by different specialists, its current status in the workflow is recorded, along with a timestamp of handoffs from one person to another. Comments made during the review process are also associated with the digitized claim. In this example, some metadata replicates information found in the document, such as the name, address, and account number. Other metadata is original content written by a person (comments) or created automatically by a computer application (workflow status). The metadata involves multiple data types, including dates (timestamps), strings (name and address), and integers (dollar amount of award).

The value of having a common metadata representation is uniform data representation, uniform access to metadata, and support for efficient searching. Uniform metadata access ensures that a common programmatic interface is available for reading and modifying metadata, and this interface doesn't vary depending on the format of the intellectual work being described (PDF, MP3, etc.). Efficient searching builds on the uniform representation of metadata, for when the data uses a common set of types, it is possible to construct the database indices that make fast searching possible.

Most computer systems that provide support for recording metadata items do so using some variation on attributes, also known as properties. An attribute pair is a name/value pair, such as *title, The Mythical Man Month*. It is akin to an instance variable in an object oriented programming language. In both cases, the data are typed and scoped to a specific instance (object instance vs. document instance). However, in programming languages the data is generally not persistent, while with attribute-value pairs it is.

The WebDAV protocol allows an arbitrary number of properties to be associated with every Web resource. A WebDAV property is a name/value pair, where the name is comprised of a namespace name (a URL or URI [2], equivalent to an XML namespace name [5]) and a property name. The value is a sequence of well-formed XML [6]. Thus, the previous example can be viewed as the triple:

(http://www.webdav.org/props/bib/, title, The Mythical Man Month)

This is represented in XML as:

<BIB:title xmlns:BIB="http://www.webdav.org/props/bib/">The Mythical Man Month</BIB:title>

WebDAV properties can be either "dead" or "live". A dead property is one whose syntax, semantics, and consistency are maintained by the client, and the server performs little, if any, processing on the data. These properties are set and updated by client applications. In the insurance claim processing example above, the comments made on each claim are an example of a dead property. If the comments have a specific format, it is up to the client application to ensure they are consistent with this format. In contrast, a live property is one where the server provides the value of the property. WebDAV's getcontentlength property is an example, since its value is a computation of the length of the resource. A live property also accommodates the case where the client provides the value for a property, and the server performs syntax and consistency checks on it. In essence, a live property is one where the server performs a computation associated with setting or retrieving its value, while a dead property has no computations beyond XML well-formedness checks.

An important class of live properties contains information necessary for the operation of the protocol. For example, WebDAV clients must be able to discover what kinds of locks are supported and currently active on a given resource. While a special purpose method could have been developed to handle this discovery, making lock discovery information available in a property allowed the metadata facilities to be reused, and reduced the total number of methods being added by the WebDAV protocol. Since many working group members felt it was important to keep the complexity of the protocol down, having fewer methods allowed the protocol to seem less complex.

WebDAV provides two methods for operating on properties, PROPFIND (read) and PROPPATCH (write). PROPFIND has three permutations for retrieving properties defined on a resource: retrieve all properties, retrieve a set of named properties, and retrieve just the property names. Retrieving all properties is useful for browsing clients that don't know the complete property set ahead of time. Retrieving a named set of properties is useful for applications that use specific metadata items. The claims processing example above is such a case, since there is a set of metadata items defined and used by the application. Finally, retrieving property names is useful for discovering whether a property has been defined on a resource.

The PROPPATCH operation submits a series of requests to set and remove properties on a resource. The entire set of requests submitted with a single PROPPATCH is executed in order, as a transaction. If a single set or remove fails, the operation reverts all affected properties back to their state at the beginning of the PROPPATCH. Due to the transactional semantics, a delete/set pair can be used to update the value of a property, since it is not possible for the property to end up in an inconsistent state if any part of the delete/set pair fairs.

Both PROPFIND and PROPPATCH can be issued to a single resource ("depth 0"), a collection and its immediate children ("depth 1"), and the entire hierarchy of a collection's children ("depth infinity"). This permits efficient retrieval and setting of properties on a large number of resources.

Experience with WebDAV Properties

Today, the most common use of WebDAV properties is for custom applications, and extensions to the WebDAV protocol. Applications of WebDAV for document management can involve the creation of custom WebDAV clients tailored to the specific use environment. These custom clients use WebDAV properties to associate document and workflow metadata with the documents being managed.

One example of this is the Extensible Computational Chemistry Environment (ECCE) developed at the Pacific Northwest Laboratory [29]. It stores a variety of chemistry-related documents and data files, using custom WebDAV properties to associate ECCE-specific metadata to stored resources. A custom client can read this metadata, and provide appropriate display. WebDAV properties have the advantage that additional properties can easily be added to existing resources by third parties. Additionally, they noted that configuring and administering a WebDAV server was easier than an OODBMS.

Another custom application example is Microsoft's Exchange 2000 email server platform. The Outlook Web Access (OWA) feature of Exchange uses the WebDAV protocol to retrieve email from a remote

Exchange server from within a Web browser running a complex browser-based email management application. Exchange defines a wide range of custom properties to model email header metadata, along with calendar events and contacts. In effect, OWA uses WebDAV as an alternative to the POP and IMAP protocols. Though the Exchange-defined properties are not standardized, other projects have been able to use them. HiPerExchange is a WebDAV server than runs on a client machine and locally caches email, thereby improving response time and disconnected email access via the OWA client [30].

Metadata-driven Web sites are another use of WebDAV properties. Since the output of PROPFIND and DASL SEARCH requests is an XML document containing the value of requested properties, it is possible to use XSLT [7] stylesheets to convert properties into HTML web pages. This permits the automatic creation of static Web sites where all or part of the content is based on property values. One example is conversion of bibliographic metadata stored on PDF documents into HTML web pages, as well as EndNote (similar to Refer) and Bibtex bibliography files. Three separate XSLT stylesheets convert a single PROPFIND/SEARCH XML output file into HTML, EndNote, and Bibtex (the latter two being textual formats).

Extensions to the WebDAV protocol have made heavy use of properties. The DeltaV protocol defines 36 new properties [8], and the access control protocol defines another 10 [9]. Properties are a convenient mechanism for protocol designers, since they provide a uniform mechanism for setting, retrieving, and marshalling protocol-specific state. Notably, the performance issues with retrieving all property names and values in a single operation has led both of these protocol extensions to exempt their new properties from this behavior.

An unexpected and disappointing outcome is the almost complete lack of support for setting WebDAV properties within authoring applications. Even though many widely used applications support WebDAV, including Microsoft Office (Word, Excel, PowerPoint), and Adobe Illustrator, Photoshop, Framemaker, InDesign, and Acrobat, they only use properties in a read-only manner. There are several possible explanations for this. Since the DASL searching protocol is not yet complete, many WebDAV repository browsers do not have the ability to search properties. As a result, even if a user could set metadata items, they have limited utility since they cannot be searched. Worse, the commonly used Web Folder client does not display WebDAV properties at all, effectively making them invisible. Despite the existence of the Dublin Core bibliographic metadata set [40], there is currently no standard representation of this information in WebDAV properties, and hence applications have no guidance on how to save these common metadata items. Finally, the lack of an explicit mechanism for setting properties upon document creation makes it impossible for a document management application to communicate a list of properties that the author should enter.

Another factor is round-tripping via the filesystem. WebDAV filesystem clients, such as Web Folders, do not locally replicate WebDAV properties when a Web resource is downloaded and saved in the local filesystem. As a result, if this resource is subsequently stored in a different location, or on a different server, the associated WebDAV properties disappear. However, any metadata stored within the document itself will survive such a round trip. Faced with this behavior, it is natural that application designers play it safe and store metadata inside the document, rather than adding support for WebDAV properties that, while stored with the document, are not stored in it. One possible solution is to have WebDAV servers act as a gateway for within-document metadata, making it available as properties.

Design Space for Metadata Storage

In the process of developing a protocol design to meet the goals for metadata support, the WebDAV working group explored a complex design space. The most significant design axes in this space are enumerated below:

- *Containment model:* the relationship between the metadata, and the primary data it describes.
- *Identifiers:* how metadata items are identified, especially when performing operations on them.
- *Typing and representation*: whether metadata is lightly or strongly typed, the kind of data types supported, and other issues concerning its representation.
- *Dynamic behavior*: metadata that is computed, or that triggers additional computation upon read/write.

- *Predefined and user contributed metadata:* the list of metadata items whose names, syntax, and semantics are predefined by the information system. Also, whether users are allowed to contribute new metadata items.
- *Metadata characteristics:* the characteristics of the metadata.
- *Schema discovery, schema update:* is it possible to discover and set information about metadata items, such as their type, syntax, ranges of allowed values, and whether they are static or dynamic.
- *Metadata operations*: which of the common metadata operations are supported, any special semantics they support, and additional uncommon or system-specific operations.
- *API packaging/marshalling:* how the logical metadata operations are grouped together into a series of concrete operations, and how they are marshaled in network messages.
- *Searching:* locating data items whose metadata matches a query.
- *Versioning:* storing the history of values of metadata items.

In general, these design axes have been chosen such that each one encompasses a set of design issues that has internal coherence and can be considered, to a large degree, independent of other design axes. Put another way, each axis represents a distinct concern in the design of metadata capabilities. As with any complex design, it is never possible to totally separate all issues into completely separate concerns, as there are always interactions among tightly coupled issues. However, by individually listing each of the primary concerns, it is possible to analyze each of these first, and then focus on their interactions, thereby simplifying the design analysis activity.

The following sections describe each of the metadata design axes in detail.

Containment Model

A key data representation question concerns whether the metadata is stored with the data it describes, as an adjunct, or whether it is stored separately and then referenced by the primary data. These two choices can be viewed as different forms of containment. The former case we term *inclusive containment* (metadata stored with the described data) while the latter is *referential containment* (metadata referenced by primary data).

Inclusion containment of metadata is far more prevalent, and typically describes an object organization where there is a primary content chunk, and then a series of metadata items. Both the content and metadata are logically part of the same object, and deletion of the object removes the content and the metadata. Older versions of the Apple Macintosh operating system (pre OS-X) exemplify the inclusion containment approach, with a file inclusively containing a data fork, holding document data, and a resource fork, holding metadata.

In the referential containment alternative, the content chunk has pointers off to the metadata items, which are typically the same kind of object as the content. Deletion of the content chunk does not necessarily affect the metadata items. The DBLP Computer Science Bibliography (dblp.uni-trier.de) demonstrates referential containment, with the bibliographic metadata for many papers pointing back, via a link, to the original article in a digital library as stored on a separate web site.

Other containment characteristics can be present, including fixed ordering among metadata items, and the possibility of cycles among referentially contained data/metadata. A complete description of these containment modeling issues can be found in [38].

Identifiers

If an information system supports metadata it provides an operation for reading that metadata, and this requires a way to refer to, or identify, the metadata to be read. There are several issues affecting metadata identifiers, including:

- Independence from (or dependence on) the described data,
- Does the identifier have any human meaning,
- Extensibility and avoiding metadata namespace collisions,
- Whether a single metadata item can have multiple identifiers.

In information systems where the metadata is inclusively contained by the data it describes, it is common to see metadata identifiers that are dependent on the data identifiers. For example, the "author" attribute on the resource "rfc2518.pdf" is completely dependent on the name of the file, since the metadata value identified by "author" varies across resources. When there is a referential containment relationship between metadata and data, each metadata instance typically has a identifier that is independent of the identifier for the data. Metadata identifiers can then be resolved without first having to resolve the primary data identifier. One example of an independent identifier is a globally unique identifier (GUID) [24] that is assigned to each metadata item instance, such as "13f81a-aac9c-8924b0c0".

The coupling of identifier types to containment relationships described above (inclusive has dependent identifiers, referential has independent ones) is not cast in concrete. It is certainly possible to conceive of assigning GUIDs to every instance of inclusively contained metadata, along with operations that retrieve metadata items identified by these GUIDs. Referential containment schemes could similarly support dependent-style identifiers, where a metadata item is retrieved based on an identifier that is relative to the described data. In this case, the retrieval function must take care of retrieving the referentially contained metadata, which in distributed systems can involve crossing system boundaries.

There are several common scenarios where a system's metadata identifiers are exposed to humans. Most information systems have a metadata browser that permits viewing of all metadata items describing a piece of data. Since the complete set of metadata items is not known in advance, the system metadata identifiers are displayed to the user. Metadata query interfaces can sometimes expose system metadata names directly to users performing queries. Additionally, developers of custom applications that work with the information system also must interact with the system metadata identifiers. Due to these scenarios, it is useful to have metadata identifiers that are meaningful to humans (like "author") instead of hard to decipher identifiers like GUIDs.

Many information systems define a core set of metadata items, and then permit this core set to be expanded by either administrators or end users. When new metadata items are defined and the identifiers are dependent, each needs a new identifier that is different from all others in use. Avoiding collisions is tricky, since typically multiple agencies wish to define compatible metadata extensions. Human readable names lead to more frequent collisions, since some metadata names, like "author," appear in multiple metadata sets. One solution is to require a central registrar of metadata names, with established policies for resolving name competition. Since central registrars require the support of a long duration human institution, avoiding central registration is often more simple than creating new institutions, or establishing registration policies with an appropriate institution, such as the Internet Assigned Numbers Authority. Pairing a human-readable metadata name with a globally unique identifier (such as a GUID, or a URI) is one way to achieve human readability and avoid namespace collisions without requiring centralized registration.

A final issue is whether a single metadata item can have multiple identifiers. If a system supports multiple similar metadata sets, there might be items that are conceptually identical across multiple sets, with each have a distinct name. For example, if a system supports both Dublin Core [40] and Refer style bibliographic metadata, the author information for a given book will be the same, but will have different identifiers (creator in Dublin Core, author in Refer). In this case, it may make sense to allow multiple identifiers for the same metadata item, so the data isn't needlessly replicated. On the other hand, this does raise the possibility of unintended side effects, since a user may delete the Dublin Core creator item and also unexpectedly delete the Refer author.

Typing and Representation

This design axis presents choices concerning how metadata items are typed, and the formats used for their representation and interchange. The first choice concerns whether metadata items are typed at all. Information systems usually have typed metadata items and use typical database data types, such as integers, floats, strings, dates, etc. However, it is possible to have only loosely typed metadata items, where all metadata is represented using strings, or a text-based format like XML. In this case, numeric values are encoded into a string in the metadata. With the XML Schema standard [3], it also possible to encode type information into XML.

The choice of typing has an impact on the persistent representation of the metadata. Metadata is often stored in a relational database, and hence strongly typed metadata allows a database to use the most efficient available data type. The binary encoding of integers and floats is far more efficient and useful than

a string encoding. It is possible for information systems to do some processing between reading the metadata from a database and returning it to clients. Hence, even if the interface between an information system and its clients passes data in a string or XML encoding, if a given metadata item is known to contain numeric data, the information system can translate between text and binary encodings as needed, and persist the numeric data in binary form.

Support for typing also affects searching, notably on supportable search operators and sort orderings. If all metadata information is in a textual format, it is impossible to perform numeric comparisons or orderings unless the search operation is given hints on how to view the metadata. In contrast, typed metadata makes it possible to perform type-specific comparisons and orderings.

For string-valued metadata, one issue is whether the metadata items have a fixed size, or, if variable sized, a fixed upper limit on their size. Usually this is governed by characteristics of the database that stores metadata, to ensure its upper storage limits aren't easily violated. This issue can also impact quota systems, which often do not account for metadata when determining space used, to improve the efficiency of this computation. Arbitrarily sized metadata can then be used to circumvent the quota system by storing large data items in metadata.

Another issue is the representation of multi-valued metadata. Consider a book with four authors. Is each author's name placed in a separate "author" item, or is there a single "authors" property containing all four names? When multiple instances of the same metadata item are permitted, it introduces additional instance and ordering parameters into operations that manipulate them. For example, if an author is subsequently removed from a working paper draft, only one author metadata item should be deleted. If the ordering of paper authors changes, the ordering of author metadata changes too. To avoid this complexity, it is possible to introduce list types, where the value is an ordered list of a given type. In the book author scenario, the four author names would be four successive items in a single metadata item with a string list type. Alternately, it is possible to use a structured value model capable of supporting arbitrarily complex values. WebDAV uses XML for property values, and hence can encode lists of values in XML element structures. Placeless Documents permits any serialized Java object to be a property value, and hence can represent any data type expressible in Java [12].

Internationalization raises additional considerations. For some human-readable metadata values, it is desirable to store multiple natural language translations of the metadata content, and then tag these metadata items with character set, character set encoding, locale, and natural language information. This ensures proper display and search sorting. While there are a wide range of issues that applications must address to perform adequate internationalization, the metadata storage facilities primarily need to provide the necessary internationalization data fields so applications can intelligently store and process multilingual metadata. Due to its impact on the value model, internationalization should be considered at initial system design time, as it may prove difficult to shoehorn rich support into an existing value model, due to the need for extra fields to store internationalization information.

Dynamic Behavior

It is often the case that metadata can be computed directly from the data, as with computing the length of a document. Some metadata can also be automatically determined by the information system, as with a last modified timestamp. Computed metadata can vary by type of document. For example, the Semantic File System supports "transducers" which compute "from", "to", "subject" and "text" metadata from email files, and procedure "imports" and "exports" on C, Pascal, and Scheme source code files [17].

Dynamically computed metadata have the benefit of being automatically maintained by the information system as well as having consistent, uniform syntax and semantics. Consider what would happen if computed metadata were unavailable, and multiple clients of an information system needed to maintain a last modified timestamp. Each client implementation would need to agree on a timestamp syntax, be consistent in their implementation of when the timestamp was updated (only when the data is modified, or also when static metadata is modified?) and maintain accurate clocks. In contrast, when the timestamp is computed automatically, the information system consistently applies a date syntax and update semantics.

In addition to dynamic computation, it is possible to perform automatic syntax and range checking of submitted values, along with a wide range of computations incorporating the value of other metadata items

and content chunks. For example, updating the available space in the quota for a directory involves summing the space used across multiple files and sub-directories.

Automatic metadata computations frequently take the form of a function, with inputs including one or more data chunks and metadata items, and the sole output being the computed metadata value. An often implicit assumption is the function does not modify the state of the information system, except to generate the output metadata value. This does not have to be the case. Reading or writing a metadata item can trigger arbitrarily complex computations that affect multiple data and metadata items. In the Placeless Documents system, metadata items can have arbitrary Java language code that is executed when operations are performed on the primary data (document) [12]. Triggering operations include adding a document to a collection, creation and deletion of metadata items, and reading the document. Code can be executed before, during, and after the execution of the operation.

It is possible to tunnel entire protocols through dynamic metadata items. One can envision writing parameters to one metadata item, then performing a read on another, thereby triggering the execution of a remote procedure. Results from the procedure are then read from yet another metadata item.

Techniques vary concerning how to update dynamically computed values. Some systems update metadata values as soon as the primary data is modified, persistently storing the results. Another approach is to batch update requests, trading off temporary metadata inconsistency for improved responsiveness on write operations. Semantic File System uses this approach, updating computed metadata on modified files every two minutes. Lazy evaluation is a third way, with dynamic metadata computation only taking place when a metadata item is retrieved or searched. Like the batched approach, lazy evaluation optimizes for write efficiency, and also ensures metadata items are consistent when retrieved. The drawback is additional complexity in read and query operations, and much slower access for the initial read of a given item. As well, lazy evaluation is inappropriate for metadata items whose value depends on the wall clock, like a timestamp.

The benefits of dynamically computed properties have led to their introduction in most software configuration management and document management systems, as well as attributed filesystems. However, these benefits have ramifications on the complexity of queries, transactions, and metadata retrieval. Many information systems use a relational database to store static metadata items, and the computed value of dynamic metadata. When static and dynamic metadata maps well into relational tables, it is easy to support queries over this metadata. However, when dynamic metadata is computed using lazy evaluation, or when the complete set of static metadata items is not knowable in advance (as when users can define arbitrary metadata items), queries become more complex because they do not map into SQL. In this case, the information system must take over some of the indexing and query plan evaluation.

Dynamic metadata can, in the extreme, make it impossible to completely roll back a transaction. Since a dynamic metadata computation can include almost any operation, it's conceivable that it can include non-reversible ones, such as transmitting a notification message over a network. If a metadata item controls access permissions, an update operation could cause the user to no longer have write permissions, and hence no longer have sufficient permission to rollback the transaction. Finally, systems such as Placeless Documents that send event notification messages upon metadata update and allow arbitrary code to be triggered by these events, make it very difficult to compute all effects of a single metadata update operation, and effectively roll back these modifications.

Predefined and User Contributed Metadata

In many information systems, there are metadata items whose names, syntax, and semantics are predefined by the system. For example, the well-known version control system RCS sets five attributes on every revision created: author, comment, timestamp, version identifier, and state [34].

Also inherent in this issue is whether the system permits the creation of metadata beyond the predefined set of metadata items. Users of RCS are not allowed to add any kind of custom metadata to versions, either static or dynamic. In contrast, the Semantic File System allows users to write new transducers that dynamically compute one or more metadata values, thereby extending the range of metadata items. WebDAV allows users to define arbitrary static metadata items, but does not provide any protocol support for creating new dynamic metadata. However, server implementations are free to implement new dynamic metadata items as they wish.

Metadata Characteristics

Metadata behavior describes general characteristics of manipulating metadata.

It is important to clearly specify whether metadata is readable, writeable, or both, and under what conditions. When metadata is used to represent data updated by the system, such as lock capability discovery metadata, it is possible that the metadata is readable, but not writeable. Other times a metadata item can only be written if certain conditions hold, for example, if a document is checked-out.

Also included in the behavior of metadata is its cacheability, the conditions under which other system components can replicate the metadata item. In general it is easy to cache static metadata, and far more complex to cache dynamic metadata. Design issues affecting caching schemes include system architecture, the kinds of dynamic metadata supported, network latency, inter-cache communication, data replication scheme, and patterns of access to the metadata.

A metadata item may be defined on only specific types of content, such as an item that is only defined on revision history objects, or on collections. The HB3/SP3 system [20], and the DeltaV protocol [8] are two examples of systems that have properties defined on specific types of objects.

Finally, there are details concerning metadata existence. Do metadata items need to have an instance on every described resource, or are metadata instances created only when desired. Can a metadata instance have a null value, or must it always have a value, with a default value used before one is explicitly set. As an example, the Nebula attributed file system has mandatory, recommended, and optional metadata items [4]. Mandatory and recommended metadata have higher consistency, since their values are either computed, or the user is forced to supply them. Optional metadata are not necessarily uniformly defined over files of a given type, and hence have lower utility.

Schema Discovery and Schema Update

The design focus here is capabilities for discovering and setting information about metadata items, such as their type, syntax, ranges of allowed values, and whether they are static or dynamic. Such information is typically termed the “schema” of the metadata, and querying the information system for metadata schema information is “schema discovery.” One pragmatic use of schema discovery allows applications to construct more meaningful displays of metadata items. For example, in the case where a numerical metadata item is transferred in textual format, such as an XML element value [6], it is helpful to know to sort it as a number. Syntax specifications and ranges of allowed values allow applications to pre-check the values of metadata items before sending them to the information system and possibly incurring errors.

The Document Management Alliance (DMA) 1.0 specification is a good example of schema discovery [11]. In DMA 1.0, every document is an instance of a specific object type, and each type specifies the metadata permitted on its instances. DMA explicitly represents object classes as objects, and hence DMA client applications can perform metadata schema discovery by retrieving data from the object representing a given class. Metadata items are characterized based on the kind of identifiers permitted, datatype, default value, singleton or enumeration, optional or mandatory, permitted values, and its search characteristics. The latter includes whether it is searchable, selectable, or orderable, and the search operators permitted.

Schema update allows applications to inform an information system about characteristics of application-defined metadata items. For example, schema update can be used to update the type of a metadata item, permitting intelligent sort ordering, and more efficient storage allocation technique. Telling the information system that a metadata item will be frequently searched permits the construction of database indices for that item, thereby improving the efficiency of searches. As the DMA 1.0 example shows, other possibilities include changing whether the property is optional or mandatory, and whether it has a default value.

One implication of schema update is the need for abstractions to represent the metadata schema information, and operations on those abstractions. This contrasts with the instance-oriented operations described below in “Metadata Operations,” since each schema update operation affects multiple metadata instances. There is also a need to go to a metadata instance and discover which abstractions are used for schema update. This affects the containment model for the metadata, since there is a need to connect the metadata instances to their schema objects, typically via referential containment.

Some schema update operations can affect the behavior of applications, sometimes breaking them. For example, changing a metadata item’s datatype from a string to an integer, or making an optional item

mandatory can easily break client applications that are hard-coded for the old behavior. On the other hand, informing an information system that a metadata item will be searched often will likely not break applications, since it's just a hint to the repository to create a search index.

Metadata Operations

Systems that support metadata tend to provide a subset of the following common operations to applications and people using the system:

Single metadata item operations:

- Retrieve a single metadata item
- Create/write/modify a single metadata item
- Delete a single metadata item
- Rename a metadata item
- Test and set single metadata item – only write the metadata item if it has a given value
- Retrieve list of metadata names that should be set upon primary data creation

Multiple metadata item operations:

- List all metadata names defined on a piece of data
- Retrieve the value of multiple metadata items
- Retrieve all metadata names and values
- Create/write/modify multiple metadata items
- Write multiple metadata items upon creation/update of the primary data
- Rename multiple metadata items
- Delete multiple metadata items

Though a seemingly straightforward set of operations, there are several issues that increase their complexity. First, is whether metadata can exist without the primary data it describes also being present. One example of this is a library catalog, where every catalog entry is separate from the media it describes.

In information systems where metadata is inclusively contained, the act of defining a metadata item can cause the creation of an empty piece of primary (document) data, since the metadata and primary data are a logical unit. The effect on dynamic metadata must be defined in this case. Creating a static piece of metadata may cause dynamically computed metadata items, such as the length of the primary data, to appear at the same time. In turn, this raises issues about the default condition of the primary data: is it defined, but with zero length, or is it undefined, with undefined content length metadata? In a similar vein, if the single metadata item that caused the creation of the primary data is deleted, then does the data retain its existence, or is it deleted as a side effect.

As an example, in WebDAV is possible to lock a null (non-existent) resource prior to writing its initial value and properties, thereby creating a "lock-null" resource. A lock-null resource has lock discovery metadata that states the kind of lock, who holds the lock, and its duration. The act of locking a non-existent resource implicitly creates it, since the lock discovery property is inclusively contained by the resource. Lock-null resources respond to GET operations with a 404 Not Found response, and hence its primary data effectively does not exist. Unlocking a lock-null resource before its primary data has been written causes it to revert back to its previous null state. Opinions vary within the WebDAV implementation community on whether lock-null resources are a good idea. All agree that they lead to significant amounts of special case code in the implementation of WebDAV operations. Some believe the added complexity is worthwhile, pointing to clients that use this capability, while others feel the benefit is too small to justify the complexity.

Systems also make different choices about the need for an explicit metadata creation operation. Often, the metadata write operation has the side effect of creating a metadata item if it was previously undefined. However, in systems where users might be manually entering the names of metadata items, this opens the possibility that a user might mistype a name, and cause the unintended creation of a metadata item. Explicitly requiring a create operation can reduce (but not eliminate) this problem, since a write operation on an undefined item will fail.

Transaction support is a thorny issue for writing and deleting multiple properties in a single operation. It is sometimes the case that dependencies exist between multiple metadata items. Since a metadata item can be viewed as an assertion about the described content, ideally either all the dependent assertions should be

updated, or none of them should. If only half of a set of dependent assertions were updated, then the entire set is potentially inconsistent, possibly making contradictory assertions about the content chunk. Requiring transactional, all or nothing behavior for the writing and deleting of metadata alleviates this problem, but increases implementation complexity, typically implying the use of transaction-capable database technology.

Transaction support for static metadata is straightforward when using a transactional database; dynamic metadata makes matters more complex. For read-only dynamic metadata, the attempt to write it causes the entire transaction to fail, and this condition can simply be checked before starting a transaction. But for arbitrary writeable dynamic metadata, the act of writing can trigger the execution of a computational process, an act that cannot be packaged into a database transaction, since the computation is taking place outside the database. It also may not be possible to roll back an arbitrary computation.

When transaction support is present, the modify metadata operation can be omitted, since it can be transformed into a delete/create pair within a transaction. This assumes that the identity of the metadata item is not meaningful, since identity would be modified by the delete. Identity might be important if metadata items have metadata defined on them, which would vanish during the delete.

An additional wrinkle on the metadata operations is whether they can be simultaneously applied to multiple objects, such as all objects in a containment hierarchy (or files in a directory hierarchy). In this case, transaction scope can be increased to have all-or-nothing behavior over all the affected objects, or transactions could be applied only to individual objects, with the possibility that metadata updates may succeed on some objects and fail on others.

Transaction support increases the complexity of error reporting as well. Since there are multiple operations that could have caused an error, ideally the exact operation that caused the termination of the transaction will be reported, including the kind of error encountered. When metadata operations can be executed across an object hierarchy, error reporting needs to identify the object affected by the error. If transactions affect just a single object, an information system may permit best effort recovery in the face of an error, allowing the system to continue applying the operation to other objects in the hierarchy. In this case, multiple errors could be reported. However, if an operation fails on all objects in a large hierarchy, it makes sense to condense the reported error to reflect this fact, rather than slavishly reporting that every individual object had the same error.

Often document management systems want the ability to specify a list of metadata items that should be set by the document author when the document is first added to the repository. Ideally, the authoring application will present a metadata entry user interface after receiving the list of desired properties. As a result, there is a need for an operation to retrieve the list of metadata that should be set upon the creation of a content object. Ideally, a system will make it possible to submit the content and metadata in a single transaction.

The operation to retrieve all metadata names and values can have significant negative performance implications in the presence of dynamic metadata. Especially for complex version control systems, calculating the value of a single item of metadata can involve multiple database interactions. Retrieving all static and dynamic properties in one operation can trigger significant amounts of disk access and computation, making this a slow operation. When a single operation can retrieve all names and values over an entire collection hierarchy, the performance hit is multiplied many times over. However, the performance implications are often not obvious to engineers using this operation, and it can be attractive to simply retrieve all names and values once, and then not worry about explicitly naming just the needed items. As a result, this operation is sometimes misused, resulting in greater than necessary system loads. A reasonable system design might omit this feature altogether, reasoning that the same effect can be accomplished by first retrieving all names, then explicitly retrieving the value of each one by name.

One way to mitigate the performance impact is to exclude certain metadata items, in effect changing the definition to “retrieve all names and values except for those which are explicitly defined to not respond to this operation.” Most dynamic metadata items can then be exempted. Of course, for some applications this behavior will be too extreme, with some dynamic metadata items wanted in the response, leading to the desire to have an “exception to the exceptions” list of metadata items that should be returned, even if they are expensive to compute.

API Packaging and Marshalling

This design axis concerns how the logical metadata operations are grouped together into a series of concrete operations. This design axis also includes how the concrete operations are assembled into network messages, and the formatting of those messages. Information systems often have a one to one mapping of logical operations to concrete operations, which makes it easy to understand which operation performs a given function. However, if transaction support is desired for the multiple metadata operations, then it makes sense to combine all metadata update/delete operations into a single concrete operation. This permits the implementation to have the sequence of update/delete operations bracketed by a transaction. For example, the WebDAV PROPPATCH method can have certain property values be set, and others deleted, all within a single transaction. Alternately, explicit begin and end transaction operations can be made available, and then any sequence of individual logical operations can be given transaction behavior. In this approach, an application would send a "start transaction" command, issue a number of update/delete commands, and when finished, issue a "finish transaction" that commits the transaction.

When sending metadata operations across a network, considerations of latency are introduced. While systems often have little control over network latency, they can mitigate its effect by reducing the total number of network round trips required to accomplish an operation. One approach is to package multiple logical operations into a single protocol operation, such as combining all metadata read operations together, as with the WebDAV PROPFIND method. Another approach is batching together multiple protocol operations into a single network message. The receiver unbatches the message, executes the operations (either individually, or as a transaction), and then batches together the responses, which are sent back in one message. NFSv4 uses this approach [32].

In its fullest complexity, the marshalling of metadata operations involves a broad range of application layer protocol design issues, including message layout, error reporting, message interaction style, underlying transport, security, and others. A full consideration of these issues is beyond the scope of this paper; we assume that these considerations have largely been addressed before starting the design of metadata operations.

Searching

One of the key benefits to creating and maintaining metadata items is the ability to search over them to find objects that match specific search criteria. Searching unleashes the value of metadata. However, the design of search languages is a complex issue in its own right, and enumerating this design space herein would detract from consideration of the other design issues involved in creating a metadata storage facility. The simplifying step of considering searching separate from metadata design is not always appropriate; the Document Management Alliance (DMA) 1.0 specification includes a cross-repository searching capability whose design did impact their metadata model [11].

Searchability is a quality metadata items may possess, describing whether the metadata item can be searched using facilities of the information system. While all metadata items can be searched by client applications reading their values and then manually evaluating search criteria against them, this is inefficient for all but small numbers of items due to the time required to retrieve them from the information system. It is often far more efficient for the information system to search metadata items, since it has low cost access to the metadata. A related notion is whether the metadata item is indexed, with such items typically permitting much faster searches than non-indexed items.

Versioning

Versioning of metadata can provide multiple benefits, including the ability to track changes and authorship to metadata items, and rolling back undesired changes. There are two main approaches to versioning metadata, depending on whether the metadata can be versioned independently of the data it describes.

When metadata items are inclusively contained by the data, information systems typically version static metadata along with the main data item. In this case, a checkout operation makes both main data and static metadata writeable; a checkin then freezes data and metadata after modifications have been made. This approach is common in Software Configuration Management systems. One permutation on this approach provides control over whether an attempt to change a metadata item causes the creation of a new revision, an example being Adele [13].

A more unusual approach to versioning metadata is to allow each versioned metadata item to have a revision history that is distinct from the data it describes. Additional operations need to be added to manipulate the versioned metadata (checkout/checkin), and existing metadata operations would require additional semantics to accommodate the fact that writeable properties now need to be checked out to be modified. In previous work we examined design issues of representing version histories [37], and do not repeat them here. While versioning of metadata was discussed at a few WebDAV design meetings, it was never incorporated into any protocol draft, primary due to its complexity.

WebDAV Metadata Design

Over the course of the development of the WebDAV protocol, three separate metadata designs were considered, each exploring substantially different aspects of the metadata design space. In a nutshell, these three approaches are (presented in chronological order):

- *Attribute Headers:* Resources inclusively contain a series of metadata items, called attribute headers, where each metadata item is a MIME typed chunk of state. Each attribute header instance has a unique URL, and existing HTTP methods GET, PUT, and DELETE are repurposed to operate on metadata.
- *Link-based Metadata:* Differentiation between "large" and "small" chunks of metadata. Large chunk metadata is an HTTP resource, and is referentially contained (using a link) by the resource(s) it describes. Small chunk metadata is stored in the type field of the link.
- *XML Properties:* Resources inclusively contain a series of metadata items, called "properties". Each property is a hierarchical tree of inclusively contained metadata elements, which are marshaled as a sequence of well-formed XML elements. New methods permit reading (PROPFIND) and writing (PROPPATCH) multiple properties at a time.

Figure 1 shows when each of these designs was considered, and the WebDAV protocol specifications that described each design.

***** Insert Figure 1 here *****

***** Insert Figure 2 here *****

Focusing for a moment on just containment aspects highlights the data model of each design. **Figure 2** shows each design's containment model, essentially an entity-relationship model where the entities are either containers or non-container atoms, and relationships are limited to inclusion or referential containment (see [19] for a more detailed discussion of the containment modeling approach).

Each metadata model has a container entity inclusively contained by the HTTP resource, which for notational convenience we call the "top-level metadata container". In **Figure 2**, the top-level metadata containers are the "attribute header", "link", and "property" entities. It is this inclusion containment relationship between HTTP resource and the top-level metadata container that associates the state of the top-level container's children with the containing resource.

The approaches all vary in the kind of children contained by the top-level container. The attribute headers design has a single value, and the MIME type of the value. The link-based design contains the link type, as well as pointers to two other resources (using URLs as pointers). Since the link type is inclusively associated with a specific HTTP resource, it has two uses, typing a link, or storing local metadata. The link-based design is the only one that uses referential containment for metadata. The XML properties approach is similar to the attribute headers, but with the addition of an extra container type (the element), representing individual XML elements. Since elements can inclusively contain other elements, this permits hierarchically structured metadata contents. While an attribute header could contain an XML formatted value, this is not explicitly part of the attribute header model.

The variation in containment among the three designs puts each one on a different trajectory through the remaining design spaces. In the following sections, each of these designs will be considered in turn, to

highlight how the containment choices influenced (or failed to have influence on) each of the remaining design choices.

Attribute Headers Design

A primary motivation for this design was the realization that HTTP already supported the return of metadata in some of the HTTP headers. HTTP headers fall into four categories, *general headers*, describing characteristics of the message being transferred, *request headers*, used to pass parameters of method calls and handle authentication, *entity headers*, used to request specific language and rendition variants of a resource as well as to describe characteristics of the response message body, and *response headers*, that provide descriptive information solely about the response message body [16]. Response headers and response entity headers provide descriptive information about the response message body, such as its last modified date, the natural language of the response, the MIME type of the response, and the entity tag, a unique identifier for the response message for that resource. In essence, these fields provide some commonly used metadata items, and their existence in HTTP headers made it desirable to have a metadata mechanism that seamlessly incorporates this header metadata.

One idea was to return all resource metadata in a series of HTTP headers returned with a GET response body, an idea used by the PICS specification [22]. However, since there can be many possible metadata items defined on a resource, the number of returned HTTP headers might be large, and unduly inflate the size of the response message. Another consideration was that user-defined metadata could be arbitrarily large, which was not the case with regular HTTP headers. Additionally, this forces every GET operation to retrieve and marshal metadata as headers, thus reducing the performance of GET. Since GET is by far the most frequently requested method, reducing its performance is not desirable, especially since most HTTP browsers are incapable of using this metadata.

However, since HTTP had set the precedent that metadata appeared in a header, a new type of "header" was needed, one that held metadata that was not transferred in a GET response. This new header type was called an "attribute header."

One marshalling design goal was to define as few new HTTP methods as possible, by reusing existing methods as much as possible. Since HTTP already had methods for reading (GET), writing (PUT), and deleting (DELETE) data, a natural question was whether these methods could be reused for metadata. A precondition for this reuse was assigning each metadata instance a unique URL, and hence each attribute header had a URL that was the concatenation of the resource URL and the property name, in the form: {resource URL}<{property name}>. Unfortunately this ignored the fact that angle brackets are illegal URL characters, since they are used as delimiters around URLs when they appear in text documents [2]. Later designs adopted different URL concatenation approaches.

However, once each attribute header had its own URL, the existing HTTP methods were defined to operate on them. This raised questions about identity. When an attribute header responds to GET, then how is it different from other resources? After all, since normal resources also respond to GET in the same way, this implies attribute headers are resources too. One way to interpret the containment model for this design is that the HTTP resource is acting as an aggregate resource, containing a set of HTTP resources within it, with one of these being the distinguished resource whose body is retrieved using GET on the URL of the aggregate resource.

This design has some interesting advantages. When GET is accompanied by one of the HTTP "Accept" headers, it specifies the retrieval of the natural language variant or alternate rendition of a resource that most closely matches the Accept header specification. This opened the possibility of reusing HTTP content negotiation for improved internationalization of metadata. With metadata retrieved using GET, it is possible to leverage the existing HTTP cache infrastructure to increase metadata retrieval performance, though this works best for static metadata. Furthermore, while this design only considered static metadata, it is conceivable that attribute values could have been the result of any arbitrary computational process used to compute a GET response body.

The reuse of existing methods had some disadvantages too, since there were no preexisting HTTP methods that could operate on multiple attribute headers at once. It also raised some awkward issues, such as the meaning of an attribute value returning an HTTP 3xx redirect message, or whether an attribute could ever require different authentication credentials than its parent.

The issues surrounding identity were impossible to quickly dispel, and led to the next design where the metadata values are first class resources, now referentially contained instead of inclusively contained.

Link-based Metadata Design

An immediate consequence of using links to point to metadata items is the need to use two protocol requests to perform logical metadata operations, such as retrieving the value of a metadata item. To read a metadata item, the destination of the link needs to be retrieved, followed by a GET on the link destination. This led to the definition of convenience methods for reading and writing the value of the resource at the destination of a link (GETLINKVAL/SETLINKVAL). These new methods can be viewed as an attempt to provide inclusion-style metadata read/write semantics for referentially contained metadata.

In the process of solving the two-round-trip issue, the new methods introduced other problems. Since two resources are involved, it is possible that the link read/write part of the operation will succeed, while the read/write on the link destination fails, without adequate mechanisms for reporting the destination errors back to the requestor. Furthermore, SETLINKVAL is not explicitly defined to be atomic, raising the possibility that only the link would be set, leaving a dangling link.

An advantage of referentially included metadata is the ability to point to metadata resources stored on a different server, such as long-lived bibliographic records stored in a digital library server. But, this is also a disadvantage. Off-server link destinations require that servers either create HTTP GET and PUT requests in order to fulfill a GETLINKVAL/SETLINKVAL, or disallow off-server destinations altogether, reducing the value of metadata by-reference.

Even when link destinations are limited to just on-server locations, metadata consistency management is troublesome. If a resource is deleted, it is difficult to construct a policy that deletes its associated metadata without requiring the server to maintain reference counts for on-server links. Worse, it is impossible for servers to know how many off-server resources may be pointing to a given metadata resource. This problem doesn't occur with inclusively contained metadata, since a delete of a resource automatically deletes all its metadata.

Using first-class resources to store metadata yields other advantages. When versioning support is provided for resources, it naturally extends to metadata too. Since every resource holds links, metadata can quite naturally have associated metadata. It also provides an elegant answer to the old question concerning the difference between data and metadata, since both are treated the same.

For metadata that is persistently stored in a database, the referential metadata approach requires a URL for each cell in the database. While feasible, it is heavyweight to have simple values like integers be their own resource, with associated URL, MIME type, and entity (however, recently the WebDAD project has explored database authoring via WebDAV, and has taken this approach [31]). One solution would be to use the "data:" URI scheme, which allows small amounts of data to be embedded in a URL [25]. The solution adopted in the link-based metadata design is to reuse the link type field to hold small data chunks when the link destination points to the resource storing the link. This led to a distinction between "small chunk" (in the link type field) and "large chunk" (the endpoint of a link) metadata, and raised the obvious problem of precisely defining the dividing line between small and large. However, introducing small chunk metadata allows typical database types to be treated as inclusively contained metadata items, including strong consistency guarantees upon delete.

Like the previous design, the link-based approach had limited support for multiple metadata operations. A metadata searching mechanism, called LinkSearch, provided support for retrieving links matching a specific type, or having a specific source or destination endpoint, and could be executed over all the resources in a directory. Hence, support exists for retrieving multiple metadata items, but not for making multiple modifications.

Naming of link types introduced the notion of embedding the schema name into the metadata item name. The design includes several predefined metadata items in the "DAV" schema, such as "DAV.SupportedLinkTypes", and even nested schemas, as with "DAV.Versioning.NextVersion".

In the end, the main problems of the design are the referential integrity and atomicity problems of linked metadata. The beneficial characteristics of linked metadata were not sufficient to overcome the desire for inclusively contained metadata, even though this design relegated it to data stuffed into the link type field.

The need for a tight integration of metadata facilities with relational database technology pushed the WebDAV protocol back to inclusively contained metadata.

XML Properties

As the WebDAV protocol was being designed, standardization work on XML was taking place within the World Wide Web Consortium. At first, this work attracted little attention within the WebDAV Working Group, partly due to ignorance of its capabilities, as well as its technical immaturity at the time. Printouts of the current XML specification were passed about during a January, 1997 meeting, and had insignificant impact on the discussion of the link-based metadata design. Over the following months XML gained considerable momentum, and by mid-year it was increasingly clear that XML was going to be a significant part of the common Web infrastructure. When the drawbacks of the link-based approach led to a reconsideration of WebDAV's property design, the working group took a gamble, and incorporated XML. This was risky since XML was not yet a proven technology, and at the time did not have a stable language specification.

In the XML properties design, metadata items return to being inclusively contained by the resource, and take on their final name, "properties". At first, an attempt was made to save some of the features of the link-based metadata model by defining a "link" XML element whose children are source and destination elements. This link element can be used in the value of a property, where the name of the property is the type of the link. However, since links are just the value of a property and have no special purpose operations supporting them, it's clear the emphasis of metadata support has shifted back to inclusively contained metadata.

Unlike previous designs, XML properties use a URI to identify the syntax and semantics of a class of properties, rather than identifying a specific property instance. A specific property instance is identified with a *resource URL, property URI* pair. Initially, the design retried URL concatenation to create a unique URL for each property instance. These concatenated URLs were constructed as follows. To avoid confusion over segment boundaries, slashes in the property URI were converted to "!". Next, the string ";DAV/" is appended to the resource URL, and to the end of this, the modified property URI is appended. For example:

http://somewhere.com/resource;DAV/(http:!!www.w3.org!standards!z39.50!author)

While syntactically a legal URL, it nevertheless appears quite strange. Worse, while individual server implementations are free to implement server-specific additions to the URL namespace (such as the ";DAV/" added above), it is much worse for a protocol to do so, since it precludes all servers supporting the protocol from using that convention for other purposes. This consideration led to property values being marshaled as the name of an XML element, using XML namespaces to represent the URI up to the final path segment [5]. Since XML namespaces distinguish between the namespace name and the XML element name, this has led WebDAV to follow suit. The complete name for a WebDAV property instance is a triple consisting of an XML namespace name (a URI), a property name (a path segment, which is used as the XML element name), and the URL of the described resource.

The underlying property value model is an inclusion containment hierarchy, where elements can contain a value, language tag, and/or child elements (see **Figure 2**, above). In the simplest case, an element can be a single data item, like an integer, and have no children. This allows a straightforward mapping of property data into relational database cells. Another case is a human readable string, where it is necessary to record the language tag, and, implicitly, the character set encoding of the string. The data model also permits properties to have a multilevel hierarchical structure. This supports multi-valued properties, such as an "authors" property that has a set of "author" values. The ability to support multiple values makes it possible to cap the number of instances of a particular property at one, thus avoiding the need to identify specific property instances.

The hierarchical element model was chosen because it directly maps to the hierarchical structure of XML elements, yielding the ability to marshal properties as sequences of well formed XML elements for transport over the network. This made it possible to combine multiple property values in a single stream of XML. The MIME multipart/related encoding is an alternative packaging mechanism, but with the perceived drawback (never well substantiated) that it is more difficult to parse. Certainly, with XML it is

possible to embed both protocol information (such as status codes) and property information together into the same XML tree, making it possible for the same tree walking code to handle both kinds of information.

XML marshalling of property data implies that each data item has an associated name, that of its containing XML element. This made it possible to skip defining a type model for properties. An underlying assumption is a client will only request a property unless it has prior knowledge of the property. As a result, information conveyed by a type, such as sort behavior, syntax, and interpretation of the value, is implicitly coded into the client, and possibly the server. So, it is not necessary to be able to discover that the "getcontentlength" property (defined in the WebDAV protocol specification) is an integer, since all client/server pairs already know this, and have encoded integer behavior. For client-defined properties that are unknown to the server, the server is unable to store the data as anything other than a string. For the base WebDAV protocol, which only supports read and writing, but not searching, this was not a major concern. If a particular property's use became widespread, servers would provide improved implementations of that property.

Ducking the definition of a property type system just delayed the inevitable. The root assumption does not always hold, since there are clients (examples are DAV Explorer [15] and kStore Explorer [21]) that retrieve all resource properties and then provide nice displays of them, including sorting of property values. The lack of type information on custom properties means that property values cannot be correctly sorted. Additionally, the DAV Searching and Locating (DASL) protocol adds server-side searching capability to WebDAV, and also needs type information to correctly sort result sets [27]. Work is currently in progress to add type information to WebDAV properties [28]. One advantage to the delay is that the World Wide Web Consortium has defined a standard for XML data types in the intervening years, making it possible for WebDAV to leverage this work, and avoid creating an incompatible set of types [3].

Properties are explicitly allowed to be dynamic or static, what the specification calls "live" vs. "dead". Live properties are permitted to exhibit a wide range of behavior, including having the server compute the value, and syntax checking of submitted values. Later work on the DeltaV protocol (which extends WebDAV with version and configuration management abilities) refined the notion of liveness, defining "protected" and "computed" properties [8]. Protected properties are not explicitly writeable by the user (i.e., by using PROPPATCH), expect via a method explicitly defined for that purpose (e.g., the LABEL method updates the label-name-set property). Computed property values are the result of a computation over the value of other properties and resources.

WebDAV properties have no indication of whether they are writeable, and even among predefined properties the write behavior varies by implementation. This has led to interoperability problems. For example, the intent of the predefined displayname property is to allow a resource's human readable name to be set by the client, and be free of some of the encoding restrictions of URLs for non-latin character sets. However, some servers force this property to have the same value as the final path segment of the resource URL, and disallow writing. The net result is clients cannot depend on this property being writeable, and the displayname capability is not used.

An advantage of using XML to marshal property data is the ability to store and retrieve Resource Description Framework (RDF) metadata in WebDAV properties [23]. This permits RDF assertions to be stored with the resource they describe, ensuring that if the resource is deleted, the assertions are removed as well. It also raises the possibility of having live properties return RDF data, ensuring consistency between resource contents and RDF assertions. Since RDF regularly embeds information in XML attributes, storage of RDF implies that a WebDAV property element also inclusively contains a set of atoms used for storing these attributes. That is, instead of having just a single atom to store the (mandated) value of the xml:lang attribute (which specifies the natural language of the XML element), there are multiple atoms to handle the possibly multiple XML attributes. The WebDAV protocol intentionally did not address whether servers are required to persistently store the value of XML attributes, since there was a deep division of opinion on this issue within the WebDAV working group. One drawback to storing XML attributes is the difficulty of storing them in a database, since each element can have an unbounded number of attributes. On the plus side, many XML technologies use attributes, and hence storing XML attributes makes it possible to leverage new work based on XML.

Early versions of this design used GET to retrieve a single metadata item, but this disappeared along with concatenated property instance URLs. PROPFIND was invented to fill the gap. With the introduction of the

PROPPATCH, then PROPFIND methods, for the first time multiple property operations were substantively addressed by the WebDAV protocol. PROPFIND combines in one method the logical operations of reading single or multiple properties, retrieving all property names, and all names and values. PROPPATCH combines the logical operations of creating and deleting individual and multiple properties into a single atomic transaction. Modification and renaming is accomplished by delete/create operation pairs. PROPFIND and PROPPATCH can be scoped to a single resource, all the resources in a collection, and all the resources in a hierarchy. The Depth header specifies the scope, and these cases are known as Depth 0, 1, and infinity.

The PROPFIND/PROPPATCH design simultaneously achieves method parsimony and good coverage of the common metadata operations, two nice qualities. There are a few drawbacks. One is that the default behavior of PROPFIND, if no message body is submitted, is to return all property names and values, thereby increasing the likelihood that this expensive property retrieval operation will be requested. Another is the overloading of PROPFIND to return the members of a collection.

Initially, an INDEX method was defined to return the members of a collection, and specific metadata about each member. A common point of debate was the exact set of metadata that should be returned. There was agreement that returning too much metadata would make the operation slow, but at the same time, it was often difficult to argue against including a specific, championed property. The resolution was to eliminate INDEX and use PROPFIND instead, allowing each application to request exactly the properties they need. However, this made it difficult to separate the operations of listing a collection's members, and retrieving collection metadata. An access restriction on the latter now affects the former.

Related Work

Many systems in the domains of Document Management, Software Configuration Management, Attributed File Systems, Digital Libraries, Hypertext, and Web Content Management provide support for setting and retrieving metadata. However, published information about these systems typically does not describe fine-grain metadata operations and their semantics. Some exceptions to this trend are the NUCM [35], Neptune [10], and Placeless [12] systems (there are likely others), along with the Document Management Alliance (DMA) 1.0 [11] and Portable Common Tool Environment (PCTE) [14] specifications. Manuals describing the application programmer interface (API) for content management systems do provide detailed descriptions of the semantics of metadata operations, but they provide no design tradeoff guidance, and these publications are often not archived. In general, the discussion in this paper is more detailed and provides more information on design tradeoffs than existing work.

In the literature on metadata, the term "metadata design" is sometimes used to denote the design of a metadata schema, rather than the system that provides persistent storage for that metadata (e.g., [41]). In this paper we do not address issues of metadata schema design. However, a specific metadata schema typically has a particular value model, and persistently storing this metadata in a system that supports a different metadata value model results in data translation. This issue was noted by [1], which handles translation explicitly using a system element called Attribute Model Translators.

Conclusion

Looking back over the WebDAV protocol design process, one question we want to answer is why it took two and a half years to produce the final specification. As the presentation in this paper has shown, many aspects of the WebDAV property design can be found in other systems. Why, then, did replicating them take so long?

There are many possible explanations. Since its membership was large, and came from a diverse set of institutional backgrounds, developing consensus in the WebDAV working group was inherently time consuming. WebDAV builds upon the HTTP protocol, itself a complex technical artifact, and one whose design was being fine-tuned during WebDAV's development. Initial inexperience among the lead designers of the WebDAV protocol also contributed to the length of development. One example of this was trying to solve too many problems at once, with early protocol drafts describing features for searching, versioning, and access control, which are each substantial problems. Though all of these factors contributed to the long development time, none of them are sufficient to completely explain it.

A more compelling explanation comes from the recognition that the WebDAV working group explored a series of substantially different design alternatives for all protocol features. Despite the familiarity of the final feature set, all of the approaches were vigorously debated, indicating disagreement about underlying assumptions. Trying to find consensus technical approaches led to exploration of design spaces for metadata, hierarchical collections, and locking, as well as initial forays into searching and versioning. One frustrating aspect of this work was the realization that others had surely explored aspects of these design spaces in the past, but nobody had adequately documented them. In the research literature, papers typically only describe a system's final form, without detailing alternatives considered, but rejected. Furthermore, systems typically encompass many aspects, and hence space dedicated any one topic is limited. The lack of a solid description of the design spaces for metadata, containment, and locking was a key contributor to the long duration of the WebDAV design activity. Its absence led to a time consuming process of developing and evaluating design alternatives, as well as discovering constraints among particular design choices.

This paper has reflected on our experience designing the property mechanism for the WebDAV protocol. A major contribution of this paper is its presentation of the metadata design space, encompassing known issues encountered by engineers in the development of metadata storage facilities for information systems. This design space considers issues of metadata identification, typing and representation, dynamic behavior, predefined and user defined metadata, schema discovery/update, operations, API packaging/marshalling, searching, and versioning. Many system designers have wrestled with this same set of issues, however none have documented their work or analyzed the tradeoffs in such detail.

The metadata design space was used to structure a retrospective analysis of the three major designs for the WebDAV protocol's metadata capabilities. Differences in containment data models among the three alternatives were shown to have a significant effect on the characteristics of each design. Deployment experience with the final design was described, highlighting that WebDAV properties are most successful in custom client/server pairs and in protocol extensions, but are not yet supported by authoring applications.

Finally, while we have tried to faithfully capture the details of the development of the WebDAV protocol's property operations, this paper is a cool-headed examination of what was an often heated and fractious protocol design activity. We take solace in Parnas' excellent justification of the activity of post-hoc design rationalization [26], and the hope that this description of design issues will help other engineers faced with the design of metadata storage capabilities.

Acknowledgements

Sections of this paper drew upon passages in the "WebDAV Book Of Why", written by the second author, available at http://www.webdav.org/papers/. Other members of the WebDAV design team, Steve Carter, Del Jensen, and Asad Faizi were also active participants in the design of WebDAV properties. The vigorous debate and review of drafts by WebDAV working group members was invaluable in developing a detailed understanding of the characteristics of the metadata design alternatives. We especially thank Larry Masinter, Judy Slein, Henrik Nielsen, Jim Davis, Roy Fielding, Geoff Clemm, Lisa Dusseault, Dennis Hamilton, Alan Babich, and Chuck Fay for their contributions towards the design. Experience with the Chimera system, and interactions with Ken Anderson and Richard Taylor refined the first author's understanding of metadata and hypertext links. Finally, the detailed comments by the reviewers were very helpful in clarifying aspects of the metadata design space.

References

[1] M. Baldonado, C.-C. K. Chang, L. Gravano, and A. Paepcke, "Metadata for digital libraries: architecture and design rationale," *Proc. International Conference on Digital Libraries*, Philadelphia, PA, 1997, pp. 47-56.

[2] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax," MIT/LCS, U.C. Irvine, Xerox Corporation. Internet Draft Standard Request for Comments (RFC) 2396, August, 1998.

[3] P. V. Biron and A. Malhotra, "XML Schema Part 2: Datatypes," World Wide Web Consortium Recommendation REC-xmlschema-2. May 2, 2001.

[4] C. M. Bowman, C. Dharap, M. Baruah, B. Camargo, and S. Potti, “A File System for Information Management,” *Proc. International Conference on Intelligent Information Management Systems*, 1994.

[5] T. Bray, D. Hollander, and A. Layman, “Namespaces in XML,” World Wide Web Consortium Recommendation REC-xml-names. January 14, 1999.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, “Extensible Markup Language (XML) 1.0 (Second Edition),” World Wide Web Consortium Recommendation REC-xml. October 6, 2000.

[7] J. Clark, “XSL Transformations (XSLT) Version 1.0,” World Wide Web Consortium Recommendation REC-xslt. November 16, 1999.

[8] G. Clemm, J. Amsden, T. Ellison, C. Kaler, and J. Whitehead, “Versioning Extensions to WebDAV,” Rational, IBM, Microsoft, U.C. Santa Cruz. Internet Proposed Standard Request for Comments (RFC) 3253, March, 2002.

[9] G. Clemm, A. Hopkins, E. Sedlar, and J. Whitehead, “WebDAV Access Control Protocol,” Rational, Microsoft, Oracle, UC Santa Cruz. Internet-Draft, work-in-progress, draft-ietf-webdav-acl-10, June 27, 2003.

[10] N. Delisle and M. Schwartz, “Neptune: A Hypertext System for CAD Applications,” *Proc. Int'l Conference on the Management of Data (SIGMOD'86)*, Washington, DC, May 28-30, 1986, pp. 132-143.

[11] DMA Technical Committee, *DMA 1.0 Specification*. Silver Spring, Maryland: AIIM International, 1998.

[12] P. Dourish, W. K. Edwards, A. Lamarca, J. Lamping, K. Petersen, M. Salisbury, D. B. Terry, and J. Thornton, “Extending Document Management Systems with User-Specific Active Properties,” *ACM Transactions on Information Systems*, vol. 18, no. 2 (2000), pp. 140-170.

[13] J. Estublier and R. Casallas, “The Adele Configuration Manager,” in *Configuration Management*, W. F. Tichy, Ed. Chicester: John Wiley & Sons, 1994, pp. 99-133.

[14] European Computer Manufacturers Association (ECMA), “Portable Common Tool Environment (PCTE) - Abstract Specification, 4th edition,” Standard ECMA-149, 1997.

[15] J. Feise, “WebDAV Explorer Home Page,” (2003). Accessed January 31, 2003. http://www.ics.uci.edu/~webdav/.

[16] R. Fielding, J. Gettys, J. Mogul, H. F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1,” UC Irvine, Compaq, W3C, Xerox, Microsoft. Internet Draft Standard Request for Comments (RFC) 2616, June, 1999.

[17] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. James W. O'Toole, “Semantic File Systems,” *Proc. 13th ACM Symposium on Operating Systems Principles (SOSP'91)*, October, 1991, pp. 16-25.

[18] Y. Goland, E. J. Whitehead, Jr., A. Faizi, S. Carter, and D. Jensen, “HTTP Extensions for Distributed Authoring -- WEBDAV,” Microsoft, U.C. Irvine, Netscape, Novell. Internet Proposed Standard Request for Comments (RFC) 2518, Feburary, 1999.

[19] D. Gordon and E. J. Whitehead, Jr., “Containment Modeling of Content Management Systems,” *Proc. Metainformatics Symposium 2002 (MIS'02)*, Esbjerg, Denmark, Aug 7-10, 2002.

[20] D. L. Hicks, J. J. Leggett, P. J. Nürnberg, and J. L. Schnase, “A Hypermedia Version Control Framework,” *ACM Transactions on Information Systems*, vol. 16, no. 2 (1998), pp. 127-160.

[21] kCura, “KStore Explorer Home Page,” (2003). Accessed January 31, 2003. http://www.kstoreexplorer.com/.

[22] T. Krauskopf, J. Miller, P. Resnick, and W. Treese, “PICS Label Distribution Label Syntax and Communication Protocols, Version 1.1,” World Wide Web Consortium Recommendation. October 31, 1996.

[23] O. Lassila and R. Swick, “Resource Description Framework (RDF) Model and Syntax Specification,”. World Wide Web Consortium (W3C) Recommendation, February 22, 1999.

[24] P. Leach and R. Salz, “UUIDS and GUIDS,”. Internet Draft (expired), <draft-leach-uuids-guids-01.txt>, http://ftp.ics.uci.edu/pub/ietf/webdav/uuid-guid/draft-leach-uuids-guids-01.txt, February 4, 1998.

[25] L. Masinter, “The "data" URL scheme,” Xerox. Internet Proposed Standard Request for Comments (RFC) 2397, August, 1998.

[26] D. Parnas, “A Rational Design Process: How and Why to Fake It,” *IEEE Transactions on Software Engineering*, vol. 12, no. 2 (1986), pp. 251-257.

[27] J. Reschke, S. Reddy, J. Davis, and A. Babich, “WebDAV SEARCH,” Greenbytes, Oracle, Intelligent Markets, Filenet. Internet-Draft, work-in-progress, draft-reschke-webdav-search-05, July, 2003.

[28] J. F. Reschke, “Datatypes for WebDAV Properties,” Greenbytes. Internet-Draft, work-in-progress, draft-reschke-webdav-property-datatypes-03, November, 2002.

[29] K. Schuchardt, J. Myers, and E. Stephan, “Open Data Management Solutions for Problem Solving Environments: Application of Distributed Authoring and Versioning to the Extensible Computational Chemistry Environment,” *Proc. High Performance Distributed Computing-10*, August, 2001, 2001.

[30] Seaside Software, “HiPerExchange: Technology Primer,” (2003). accessed February 1, 2003. http://www.seasidesw.com/resources/product_datasheets/HiPerExchangeTechnologPrimer.pdf.

[31] B. Shagdar and I. Holyer, “WebDAD: A WebDAV Implementation for Authoring Databases,” *Proc. IADIS International Conference WWW/Internet 2002*, November, 2002, pp. 827-828.

[32] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, “NFS version 4 Protocol,” Sun Microsystems, Hummingbird, Zambeel, Network Appliance. Internet Request for Comments (RFC) 3010, December, 2000.

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

[34] W. F. Tichy, “RCS - A System for Version Control,” *Software-Practice and Experience*, vol. 15, no. 7 (1985), pp. 637-654.

[35] A. van der Hoek, “A Testbed for Configuration Management Policy Programming,” *IEEE Transactions on Software Engineering*, vol. 28, no. 1 (2002), pp. 79-99.

[36] WebDAV Working Group, “WebDAV Distributed Authoring Protocol Version History,” (1999). accessed January 30, 2002. http://www.ics.uci.edu/pub/ietf/webdav/protocol/.

[37] E. J. Whitehead, Jr., “Design Spaces for Link and Structure Versioning,” *Proc. Hypertext 2001, The Twelfth ACM Conference on Hypertext and Hypermedia*, Århus, Denmark, August 14-18, 2001, pp. 195-205.

[38] E. J. Whitehead, Jr., “Uniform Comparison of Data Models Using Containment Modeling,” *Proc. Hypertext 2002, The Thirteenth ACM Conference on Hypertext and Hypermedia*, College Park, MD, June 11-15, 2002.

[39] E. J. Whitehead, Jr. and Y. Y. Goland, “WebDAV: A Network Protocol for Remote Collaborative Authoring on the Web,” *Proc. Sixth European Conference on Computer Supported Cooperative Work*, Copenhagen, Denmark, Sept. 12-16, 1999, pp. 291-310.

[40] S. Wiebel, J. Kunze, C. Lagoze, and M. Wolf, “Dublin Core Metadata for Resource Discovery,” OCLC Online Computer Library Center, UC San Francisco, Cornell, Reuters. Internet Information Request for Comments (RFC) 2413, September, 1998.

[41] C. C. Yang and W. W. M. Chan, “Metadata Design for Chinese Medicine Digital Library Using XML,” *Proc. 33rd Hawaii International Conference on System Sciences*, 2000.

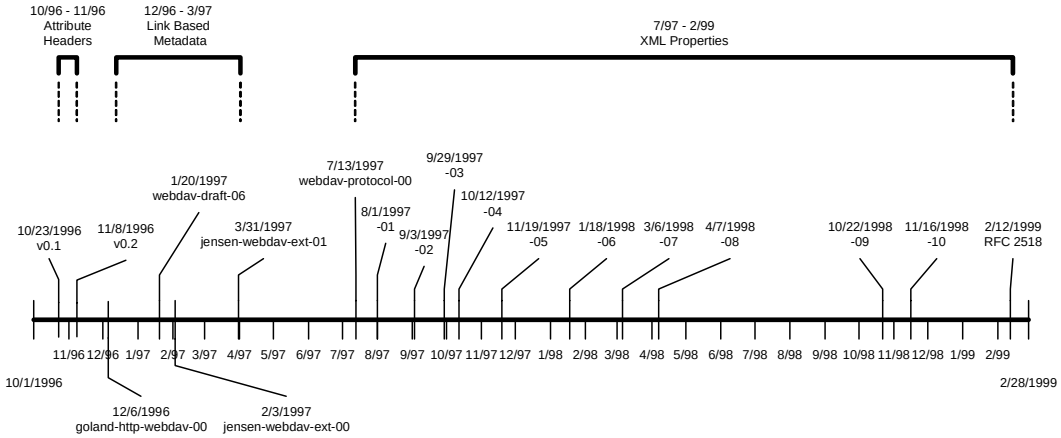

Figure 1 – Timeline showing the publication date of individual WebDAV protocol drafts (bottom), and the timespans for successive metadata designs (top). Consistent naming and version numbering of WebDAV specifications did not begin until draft-ietf-webdav-protocol-00, released in July, 1997. While all of these drafts are now expired IETF Internet Drafts, and hence no longer available from the IETF, archived copies can be accessed via [36].

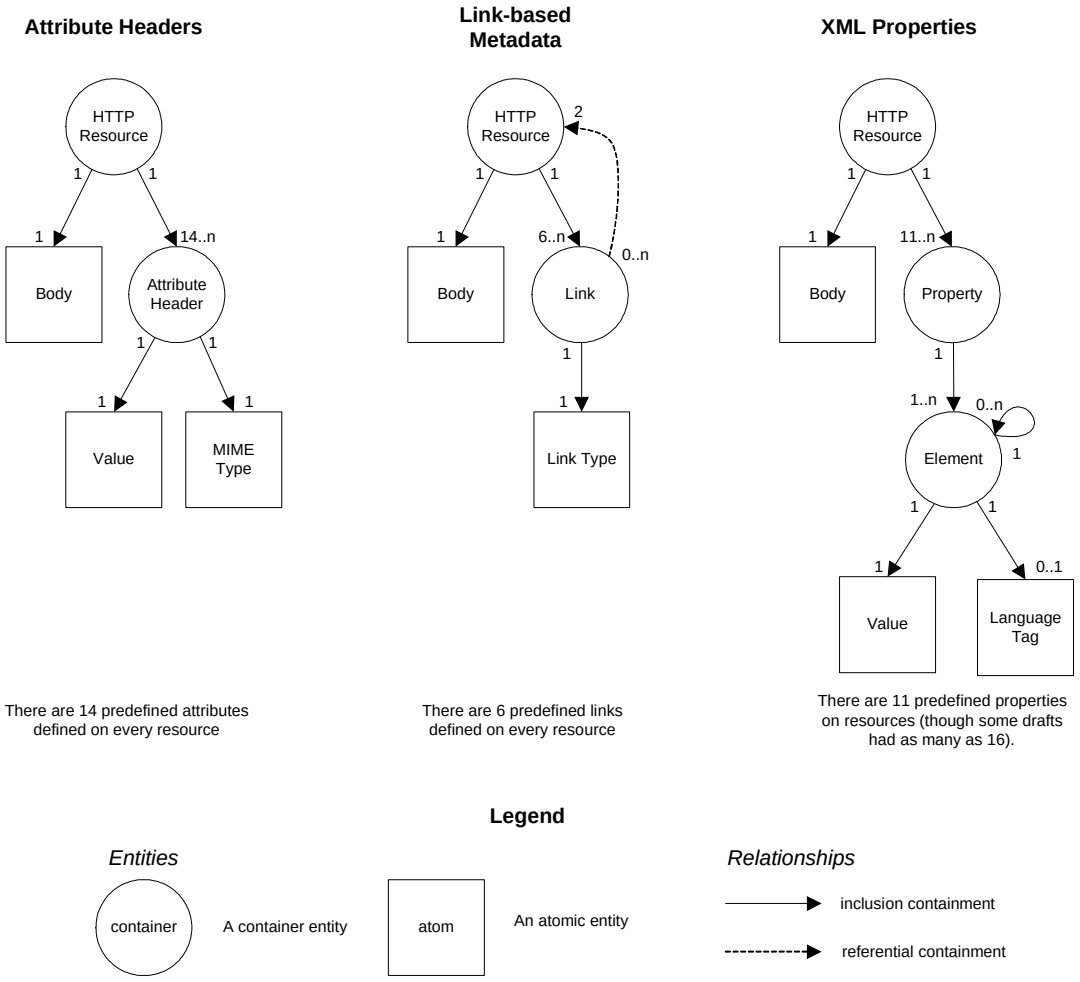

Figure 2 – Containment models of WebDAV metadata designs.