INTERNET-DRAFT
draft-ietf-webdav-version-goals-01.txt

Jim Amsden, IBM
Chris Kaler, Microsoft
J. Stracke, Netscape

Expires December 26, 1999   June 26, 1999

Goals for Web Versioning

Abstract
Versioning and configuration management are important features for controlling the evolution of remotely authored Web content. Parallel development leverages versioning capability to allow multiple authors to simultaneously author Web content. These functions form a basis for flexible, scaleable distributed authoring. This document describes a set of scenarios, functional, and non-functional requirements for web versioning extensions to the WebDAV protocol. It supersedes the versioning-related goals of [WEBDAV-GOALS].

Table of Contents

1 INTRODUCTION

Versioning, parallel development, and configuration management are important features for remote authoring of Web content. Version management is concerned with tracking and accessing the history of important states of a single Web resource, such as a standalone Web page. Parallel development provides additional resource availability in multi-user, distributed environments and lets authors make changes on the same resource at the same time, and merge those changes at some later date. Configuration management addresses the problems of tracking and accessing multiple interrelated resources over time as sets of resources, not simply individual resources. Traditionally, artifacts of software development, including code, design, test cases, requirements, help files, and more have been a focus of configuration management. Web sites, comprised of multiple inter-linked resources (HTML, graphics, sound, CGI, and others), are another class of complex information artifacts that benefit from the application of configuration management.

The WebDAV working group originally focused exclusively on defining version management capabilities for remote authoring applications and group consensus on these features is reflected in [WEBDAV-GOALS]. However, as the WebDAV working group has constructed protocols for versioning functionality, it has become clear that while versioning functionality alone is useful for a range of content authoring scenarios involving one, or a small set of resources, versioning alone is insufficient for managing larger sets of content. Protocol support for parallel development and simple remote configuration management of Web resources provides functionality for managing larger sets of interrelated content developed by multiple users at different locations.

This document contains a set of scenarios and a list of the functional and non-functional goals for versioning, parallel development, and configuration management of Web resources. It replaces the existing functional goals for versioning capability described in [WEBDAV-GOALS], section 5.9. These scenarios and goals are used to develop a model of WebDAV versioning, which in turn is used to develop the protocol that implements it.

Version management is always a tradeoff between the goals for maximum data integrity, maximum data availability, and ease of use. It is relatively easy to specify a design that satisfies any two of these goals, but this is often at the expense of the third. For example, data availability and ease of use are easy to accomplish using authoring servers that compromise data integrity by following a last writer wins policy. In contrast, high data integrity and availability are possible using branch and merge systems, but at the cost of ease of use due to difficult merges. The requirements for WebDAV versioning are based on compromises between these conflicting goals. WebDAV versioning specifies a set of mechanisms that can be exploited to support a variety of policies allowing client applications and users to find a balance appropriate to their needs.

1.1 Definitions

1. A basic resource is a resource that is not a collection or reference, i.e., an HTTP/1.1 resource.

2. A versioned resource is an abstraction for a resource which is subject to version control, a resource having a set of revisions, relationships between those revisions, revision names, and named branches that track the evolution of the resource.

3. A revision is a particular version of a versioned resource. An immutable revision is a revision that once created, can never be changed without creating a new revision. A mutable revision is a revision that can change without creating a new version.

4. A working resource is an editable resource derived from a revision of a versioned resource by checking out the revision. A working resource can become a new revision, or overwrite an existing mutable revision on check in.

5. A initial revision is the first revision of a versioned resource and has no predecessors within the versioned resource.

6. A revision name is a unique name that can be used to refer to a revision of a versioned resource. There are two types of revision names, revision identifiers or labels as described below.

7. A revision identifier (or revision ID) is a revision name which uniquely and permanently identifies a revision of a versioned resource. Revision identifiers are assigned by the server when the revision is created and cannot be changed later to refer to a different revision.

8. A label is a revision name which uniquely, but not necessarily permanently identifies a revision of a versioned resource. A label may be assigned to a revision, and may be changed to refer to a different revision at some later time. The same label may be assigned to many different versioned resources.

9. A predecessor of a revision is a revision from which this revision is created. A successor of a revision is a revision derived from this revision. A revision may have one predecessor and multiple successors. The is-derived-from

relationships between revisions of a versioned resource form a tree.

10. The merge-predecessors of a revision are those revisions that have been merged with this revision.

11. A revision history is a concrete representation of the elements of a versioned resource including all predecessor and successor relationships, revision names, activities, etc.

12. A line-of-descent is a sequence of revisions connected by successor/predecessor relationships from the initial revision to a specific revision.

13. An activity is a resource referring to a named set of revisions that correspond to some unit of work or conceptual change. Activities are created by authors and are used to organize related changes to resources, and to provide a basis for parallel development and merging concurrent changes to the same resource. An activity can contain revisions of multiple versioned resources, and/or multiple revisions of the same versioned resource along a single line-of-descent. In each activity, it is possible to refer to the latest revision of a versioned resource in that activity.

14. A workspace is a resource that is used to determine what revision of a versioned resource should be accessed when the resource is referenced without a particular revision name. When a user agent accesses a versioned resource, a workspace may be specified to determine the specific revision that is the target of the request. A workspace contains a version selection rule that is applied when the workspace is used in conjunction with the URI for a versioned resource to perform URL mapping and select a specific revision.

15. A revision selection rule specifies what revision of a versioned resource should be selected. WebDAV defines selection rules that allow a revision to be selected based on its checked out status, revision name, activity name, configuration name, or the latest revision. Servers may support additional selection rules.

16. A conflict report lists all revisions that must be merged when an activity is merged into a workspace. If the merge source activity specifies a resource that is a predecessor or successor of the revision selected by the current workspace, then there is no conflict. The merged workspace will pick the revision already in the workspace if the merge source specifies a predecessor, otherwise it will pick the successor specified by the merge source. Conflicts result when the merge source activity picks a revision on a different line-of-descent than that selected by workspace. Conflicts are resolved by merging resources together into the workspace. This creates a new revision that has multiple predecessors and

contains the changes from both merge source and the current workspace revisions.

17. A configuration is a named set of related resources where each member refers to a specific revision of a versioned resource. A configuration is a specific instance of a set of

versioned resources. Configurations are similar to activities, but play a different role. A workspace with its current activity and version selection rule specifies what a client can see. An activity is associated with work in progress and encapsulates a set of related changes to multiple versioned resources. Creating separate activities allows developers to work in parallel on the same resources, and to reconcile conflicts through merging activities. Configurations represent a persistent selection of revisions of versioned resources for organization and distribution. Configurations can be versioned resources, activities cannot.

18. The checkout paradigm is the process by which updates are made to versioned resources. A resource is checked out thereby creating a working resource. The working resource is updated or augmented as desired, and then checked in to make it part of the version history of the resource.

1.2 Storyboards

This section provides an example usage scenario that provides a context for explaining the definitions above, and for exploring and validating the goals given in the rest of this document. The example consists of a fictitious company, Acme Web Solutions that is developing a typical Web e-business application. To provide for the broadest coverage, the scenarios start with a non-existent resource typical of web applications, and follow its life cycle through development and multiple deployments. Other resources would likely have similar life cycles.

Acme Web Solutions (AWS) has developed a web-grocery store called WGS. The application consists of a number of HTML pages, some Java applets, some Java Server Pages (JSP) and a number of Java servlets that access a DB2 database.

AWS has decided to develop a new generation of its flagship WGS product to include maintenance of customer profile information, and active (push) marketing of product specials to interested customers using Channel Definition Format (CDF). The new product will be called Active Grocery Store or AGS. Customers who are interested in receiving information on specials will indicate that interest by subscribing to various CDF channels targeting pre-defined or user-specified product groupings. Since AGS represents significant new revenue potential for grocery stores, AWS has decided to sell it as a separate product from WGS, and at a relatively high price. WGS will still be available without AGS as a lower-cost, entry-level solution for smaller stores, or stores just getting into e-business solutions.

AGS is a typical Web application development project that will require changes to existing resources in AWS as well as adding new resources. These new resources will also be HTML pages, applets, JSPs, servlets, etc. WGS is an active project sold to current customers with a maintenance contract. It has on-going updates that are unrelated to the new AGS system, but may need to be included in the AGS system. These include bug fixes or minor new functional improvements. Since AGS is based on WGS, but both can evolve and be sold separately, it is necessary to maintain versions of resources used by both. This will require AWS developers to specify a configuration of versioned resources corresponding to each product. As the products evolve over time, these configurations will be versioned resources themselves, each representing a new release of their associated product, WGS, AGS, or both.

The AWS development organization consists of a large number of developers across a variety of disciplines including webmasters, Java developers, relational database developers, HTML page editors, graphics artists, etc. All of these developers contribute to the development of the WGS and AGS products, often working in parallel on the same resource for different purposes. For example, a WGS developer may be editing an HTML page to fix a usability problem while an AGS developer is working on the same page to add the new AGS functions. This will require coordination of their activities to provide maximum availability of these shared resources while at the same time ensuring the integrity of the updates. The AWS development team has decided to allow parallel development and resolve multiple concurrent updates through branching and merging of the resource version graph. This adds complexity to the development project as well as some risk due to inaccurate merges, but AWS has decided it cannot be competitive in the Web world if all development must be serialized on shared resources as this would significantly slow product development.

The following scenarios trace the life cycle of a typical Web resource from conception to product deployment and maintenance. Each scenario exposes some aspect of WebDAV and its use of the versioning, parallel development and configuration management definitions and goals specified in this document. In the scenarios below, it is assumed that all developers have access to a Web WorkBench (WB) application that provides client access to a WebDAV server called DAVServer. It is further assumed that both the client and server provide level 2 WebDAV services plus advanced collections, versioning, parallel development, and configuration management.

There is a goal that WebDAV versioning will support perhaps multiple levels of versioning from none (existing WebDAV specification), simple linear versioning, support for parallel development, and through to configuration management. The scenarios below should follow this progression from simple to complex in order to help expose logical points for leveling the protocol functionality. However, the intent of this document is to at least expose the complete goals for full WebDAV versioning support in order to ensure down-levels are a consistent subset. The exact contents of down-level servers and the number of levels will be determined later during protocol development.

1.2.1.1 Resource Creation

The AGS project team held a design meeting to determine the work products required to support the AGS project, its integration with the WGS application, and to assign these work products to developers. Various analysis and design techniques can be used to discover the required work products, but this is beyond the scope of WebDAV. At the end of the meeting, webmaster Joe was assigned to develop the new welcome page, index.html, for the AGS project. This page will be the initial page used to navigate the AGS application, and is the first page seen by users. It is a new page that will not replace the WGS welcome page, but will contain a reference to it.

Joe uses WB to create a new collection, http://aws/ags/, and the new index.html page in the collection http://aws/ags/index.html. Neither the parent collection, nor index.html are versioned resources at this point. A WebDAV MKCOL is used to create the collection, and a PUT is used to create the initial, empty resource.

1.2.1.2 Resource Editing

Joe uses WB to GET the resource and edit it with his favorite HTML editor. Each save by the HTML editor does a PUT to the DAVServer, overwriting its current contents. No new versions are created. Joe may also use WB to get and set properties of index.html using PROPFIND and PROPPATCH. Joe does not need to lock index.html because he is the only developer working on it at this time. He could however lock the resource to ensure no one else could make any changes he is not aware of.

1.2.1.3 Creating a Versioned Resource

At some point, Joe decides preliminary editing on index.html is complete, and he needs to make a stable version available to other developers who need it for integration testing, etc. Joe however wants to ensure that no other developers make changes to index.html that he cannot back out, as he is the webmaster responsible for the resource. So Joe uses the WB to make index.html which causes DAVServer to create a versioned resource, and make the initial version Joe's index.html. At this point, Joe's index.html is immutable, it cannot be changed by anyone, including Joe, and remains in the repository until the versioned resource is deleted.

1.2.1.4 Labeling a Version

When DAVServer created the versioned resource corresponding to index.html, it gave the initial version a revision id, "102847565". This revision name is automatically assigned by the server, and cannot be changed or assigned to any other version. This revision name acts as the unique identifier for this version of versioned resource index.html. The AGS development team has decided that a revision label _initial_ will identify the initial version of all resources. This ensures they stand out and can be easily accessed without remembering some opaque revision id. Joe uses WB to set the label on the initial version to "initial" in order to identify the version with this more meaningful name.

1.2.1.5 Accessing Versioned Resources

Fred wants to access Joe's initial version of index.html. So he uses URL http://aws/ags/index.html to get the contents of the resource and notices he does get the right version, because it was selected by the default workspace. That is, when Fred accessed URL http://aws/ags/index.html, he did so without specifying a workspace. So the default workspace was used, and the default workspace always uses "latest" in its version selection rule. But Fred wants to be more cautions. He wants to be sure that he continues to get version labeled "initial", even if the latest version changes as the result of new changes Joe may check in. So Fred creates a workspace called "initialws", and sets the version selection rule to be the revision labeled "initial". Then Fred always accesses index.html with its URL and the initialws workspace to be sure he gets the specific version he needs. The workspace also ensures he gets the revision named _initial_ of all other versioned resources as well, ensuring a consistent set of revisions.

Later that week, there have been a number of changes to index.html, and Fred wants to just take a quick look at an old version to remember how the page used to look. Fred's workspace is currently selecting the latest version, and he doesn't want to change his workspace just to look at some other revision. So Fred uses his WebDAV client to access index.html using label _initial_, or revision id 32345 to override the workspace selection and get the initial revision.

1.2.1.6 Creating a New Revision

A week later, a number of developers have noticed that index.html is missing both important references to their pages as well as hot images for navigation. They send email to Joe specifying their new requirements. Joe now wants to make changes to index.html and create a new revision. He wants to retain the old revision, just in case the requirements he was given were incorrect and need to be backed out, and to allow developers using the old revision to continue their work. To do this, Joe uses the WB to check out index.html and create a new working resource. Joe can now access the working resource because working resources are always visible from the workspace in which they were checked out.

As before, Joe uses the WB and HTML editor to GET the working resource and PUT updates. Each PUT replaces the contents of the working resource with changes made by the HTML editor, no new revision is created. When Joe is finished making edits to support the new requirements, he checks the working resource back in, making a new revision.

1.2.1.7 **Editing a Mutable Revision**

John was assigned to write a high level marketing document, ags.html that provided an overall description of the AGS application. Since most changes to this document have no effect on the rest of AGS, John decides to allow revisions of ags.html to be overwriteable. This is so simple spelling and grammar errors can be fixed without requiring the creation of a new revision. John still wants to create revisions whenever some significant new feature is added to AGS so the old descriptions are available to customers who don't upgrade.

John creates resource ags.html, edits it a number of times, and then checks it in to create a versioned resource.

Later on, a new feature is added and John checks out ags.html to create a new revision, makes his edits, and checks it back in, creating a new revision. Three days later, John notices a spelling mistake in the first revision that he corrected in the new revision, but users of the old revision would like the correction made for their users too. So John again checks out the old revision creating a new working resource, fixes the spelling mistake, and then checks the working resource back in. However in this case, John selects check in in-place in order to overwrite the old revision with the corrected revision. Now all users of the old revision will see the correction. This revision is now marked as mutable since it has been changed.

Six months later, there have been a number of complaints about ags.html presenting misleading product information that has resulted in unhappy customers. There's even talk of lawsuits. So John hurriedly updates ags.html and checks in the new version as immutable so that in case there is a suit, he can prove that customers had access to his updated version. Now any changes can be made by creating new immutable revisions without ever worrying about loosing old version.

A year later, things have cooled down, and John decides its OK to allow mutable revisions again. On his last change he checked ags.html in as a mutable revision allowing subsequent changes to be done without creating new versions. At the same time, the revision history of the immutable revisions is preserved just in case that pesky customer re-appears.

1.2.1.8 Parallel Development with Activities

Two weeks later, there is a major redesign of AGS that results in a lot of changes to index.html. Again, Joe checks out the resource creating a new working resource. But it is taking Joe a long time to finish all the edits, and in the meantime, graphics artist Jane wants to update index.html with references to the new images that resulted from the AGS redesign. Jane attempts to check out index.html, but WB informs her that Joe already has it checked out and refuses the request. She checks with Joe, and since they are both working on different aspects of index.html, Joe feels it would be fine for Jane to do her work in parallel with his, and then he will merge her changes with his to finish the required updates. Jane creates a new activity called "images_updates", uses it to set the activity of her workspace, and again attempts the checkout. This time the checkout succeeds, and a new working resource is created for index.html in the images_updates activity. Now any changes that Jane makes to images.html are completely independent of changes Joe makes to the same resource, but in a different activity. Note that Joe did not create an activity when he checked out index.html. Instead, the default activity "mainline" was used. Jane couldn't checkout index.html without specifying a different activity because a resource can only be checked out once in a given activity. She also couldn't make any changes until the resource is checked out as checked in revisions are read-only.

After making her edits, she checks index.html back in, which creates a new revision in the images_updates activity.

1.2.1.9 Merging Activities

Project management practice dictates that at various times during the development project, usually every few days or at specific project milestones, the updates from any parallel activities should be merged in order to integrate the changes and produce instances of the products suitable for testing. This avoids the risk of revisions of shared resources diverging wildly, and thereby decreases the likelihood of difficult or inaccurate merges. It also encourages communication within the development organization and avoids "big-bang" integration points late in the development cycle. This enhances the stability of the products and helps ensure a deterministic, controllable development process. It also allows early product testing and better feedback to developers.

Joe has finally finished his changes to image.html, and is ready to incorporate the changes from Jane's images_update activity to get the new images. Before doing so, Joe checks his updates into revision "r0.2" so if he does something wrong when doing the merge, he can recover and try again. Now Joe specifies in his workspace that he wishes to merge the "image_updates" activity into his workspace. He then can obtain a conflict report from his workspace that indicates that the resource index.html requires a merge. He then issues a merge request for index.html. This checks out the resource in the mainline activity (the activity in Joe's workspace), and registers a merge from the latest revision in the image_updates activity to the working resource. This working resource now has two predecessors, r0.2 and the image_updates revision. Joe then uses the differencing capability in his HTML editor to find the differences between his revision and Jane's, and to apply Jane's changes as appropriate.

The HTML editor Joe uses is WebDAV versioning aware, and does a 3-way merge by accesses the closest common ancestor in the revision history in order to help with the merge process. Joe notices that most of Jane's changes do not conflict with his as they are in different places in the resource, but there are a number of places where he added new functions that do not have images as Jane didn't know they were there. He notes these and either fixes them himself, or sends email to Jane so she can fix them in another revision. Once the changes are complete, Joe checks in the merged revision. Jane is free to continue making updates in her image_updates activity, and these changes can be merged in again later.

1.2.1.10 Creating a Configuration

At some point, enough of the work products of the AGS application are sufficiently complete and stable that AWS wants to distribute an alpha release. To do this, Joe uses WB to create a configuration named "alphaRelease" that will contain a consistent set of compatible work product revisions. This configuration will contain all revisions currently selected by Joe's workspace. If any working resources exist in Joe's workspace, the request to create a configuration fails, with an error message indicating that the failure is due to the presence of checked-out resources in Joe's workspace.

When Jane is ready to see the alphaRelease, she modifies the revision selection rules of her workspace to select this new configuration. Any conflicts between this new configuration and her current activity requiring merges would be noted in the "conflicts" report of her workspace, which Jane could then resolve with the "merge" operation.

Each release of AGS consists of new resources and updated revisions of existing resources. To simplify creating a new configuration for each new release, Joe can make the AGS configuration a versioned resource. For release 1 of AGS, Joe uses a configuration called AGS, and labels it R1. For release 2, he checks out version R1 of configuration AGS, and adds, removes, or changes the revisions of versioned resources in the configuration, then checks in the configuration and labeling it R2.

1.2.1.11 Getting the Revision History of a Versioned Resource

In order to determine what revision should be included in the alphaRelease configuration, Joe must examine the revision history of resource index.html. He does this by requesting the revision history of index.html and receives an XML document describing all the revisions including their revision id, labels, descriptions, successors, predecessor, and merge predecessors. Joe uses an XML enabled browser and an XSL style sheet to view the revision history.

1.2.1.12 Accessing Resources by Non-versioning Aware Clients

Fred belongs to a different company, and does not have any WebDAV versioning aware tools. However, he is an excellent graphics artist, and has been asked to look over a particular image file, logo.gif. So Fred uses his image editing tool to get a copy of logo.gif. Because his editing tool is not versioning aware, he cannot specify a particular version, either with a revision name or by using a workspace. However, the WebDAV server provides a default workspace that selects the latest revision when no label or workspace is specified on a request.

1.2.1.13 Updating Resources by Non-versioning Aware Clients

Fred has provided his review to Jane and Joe, and they decide he should be allowed to update the image in logo.gif. Fred then edits the image in his image editing tool, and attempts to save it on the DAVServer. Again, the editing tool does not specify a workspace, or activity, nor can Fred check out the resource before attempting the save. Joe realizes Fred must be able to change the resource, so he enables automatic versioning in logo.gif. Then when Fred attempts to update the resource, the server automatically checks out the resource, does the put, and then checks it back in, all in the context of the default workspace.

If someone else had the resource already checked out, then Fred's save would have failed because the automatic check out would have failed.

There are some potential problems with using non-versioning aware clients this way. If Fred got a copy of the resource, and then Jane checked it out, made changes, and then checked it back in, when Fred does his save, Jane's changes will be lost. The changes will appear in a previous revision, but they may have been in the same activity, and there would be no indication that a merge needs to be done in order to pick up both changes. To avoid this problem Joe could change the activity in the default workspace so that all changes done by non-versioning aware clients are done in a separate activity. This would allow Joe to control when these changes were merged back into other activities.

1.2.1.14 Freezing an Activity

Joe has decided that the imageUpdates activity should no longer be used once all the changes in that activity have been merged into the mainline activity. To enforce this, Joe locks the activity. Then when Jane attempts to edit index.html in her imageUpdates activity, the checkout fails as the activity is locked.

1.2.1.15 Preventing Parallel Development

Joe is responsible for another resource, getPreferences.shtml that he wants complete control over. He does not want to allow anyone else to ever make changes to this resource in any activity. To enforce this, Joe indicates getPreferences.shtml does not support multiple activities, and he checks it out to make sure no-one else can make any changes. Then when Jane attempts to checkout getPreferences.shtml in the imageUpdates activity, the checkout fails indicating that resource does not support parallel development.

1.3 Goals

This section defines the goals addressed by the protocol to support versioning, parallel development, and configuration management. These goals are derived from the desire to support the scenarios above. Each goal is followed by a short description of its rationale to aid in understanding the goal, and to provide motivation for why it was included.

1. Versioning aware and non-versioning aware clients must be able to inter-operate. Non-versioning aware clients will not be able to perform all versioning operations, but will, at a minimum, be capable of authoring resources under version control and be capable of creating new revisions while implicitly maintaining versioning semantics. Non-versioning aware clients are HTTP/1.1 and versioning unaware WebDAV clients.

   Versioning and configuration management adds new capabilities to WebDAV servers. These servers should still be responsive to non-versioning aware clients in such a way that these clients retain their capabilities in a manner that is consistent with the versioning rules, and the capabilities those clients would have had on a non-versioning server. For example, non-versioning aware clients should be able to GET the contents of a versioned resource without specifying a revision and get some well-defined default revision. A non-versioning aware client should be able to PUT to a versioned resource and have a new revision be

automatically created. The PUT must be done by doing an implicit
checkout, PUT, and checkin in order to maintain versioning
semantics and avoid lost updates. A subsequent GET on the same
versioned resource by this client should return the new revision.
The server should be able to be configured so that these non-

versioning aware client updates are placed in a different activity, or perhaps disallowed.

2. It must be possible to version resources of any media or content type.

   The versioning semantics of the protocol must not depend on the media type of the resource or versioning would have limited applicability, and client applications would become more complex.

3. Every revision of a versioned resource must itself be a resource, with its own URI.

   See section 5.9.2.2 of [WEBDAV-GOALS]. This goal has two motivations. First, to permit revisions to be referred to, so that (for example) a document comparing two revisions can include a link to each. Second, revisions can be treated as resources for the purposes of DAV methods such as PROPFIND.

4. It must be possible to prevent lost updates by providing a protocol that reserves a revision of a resource while it is being updated and preventing other users from updating the same revision at the same time in uncontrolled ways.

5. It must be possible to reserve the same revision more than once at the same time, and to have multiple revisions of the same versioned resource reserved at the same time.

6. It should be possible for a client to specify meaningful labels to apply to individual revisions, and to change a label to refer to a different revision.

   Although the server assigns unique revision IDs, human-meaningful aliases are often useful. For example, a label called "CustomerX" could be assigned to the latest revision of a document which has been delivered to customer X. When X calls to inquire about the document, the author(s) can simply refer to the label, rather than maintaining a separate database of which revisions have been shipped to which customers.

7. It must be possible to use the same label for different versioned resources.

   This allows authors to indicate that revisions of different resources are somehow related or consistent at some point in time. Configurations formalize this relationship.

8. The labels and revision IDs within a revision history are names in a common namespace, in which each name must be unique. The

server may partition this namespace syntactically, in order to
distinguish labels from IDs. The server enforces uniqueness for
these labels.

This means the same label cannot apply to multiple revisions, the same revision ID cannot apply to multiple revisions, and no label can also be a revision ID or vice versa. This is required so that a label, when applied to a versioned resource, refers to one and only one revision, and all revision names for a versioned resource are unique. To enforce uniqueness, a server will have to reject labels that it might eventually use as revision IDs. The simplest way to do this is to partition the namespace.

9. Given a URI to a versioned resource, and a revision name, it must be possible for a client to obtain a URI that refers to that revision, and to access the revision.

   This allows specific revisions of a resource to be accessed given the URI of the versioned resource and a revision name.

10. Given a URI to a versioned resource, and a workspace, it must be possible for a client access the revision selected by the workspace.

    When a user agent accesses a versioned resource, it is necessary to provide additional information to specify which revision of the versioned resource should be accessed. One way to do this is to specify a revision name with the resource URL to select a particular revision as specified in the previous goal. However, this requires users to add and remember a label for each revision, which is inconvenient and does not scale. In addition, labels alone don't provide a way of accessing revisions modified in an activity, or contained in a configuration. It is possible to specify a number of different ways of accessing specific revisions using different headers for labels, activities, configurations, working revisions, etc., but this leads to a lot of complexity in the protocol, and for users. Workspaces provide a unified means of specifying how URLs are mapped to specific revisions. A workspace contains a revision selection rule that is applied when the workspace is used in conjunction with the URLs for versioned resources to perform URL mapping to select a specific revision. This allows specific revisions of a many, related revisions to be accessed through URLs without having to specify a specific label for each resource. It also provides a means to resolve URLs to particular revisions using more complex revision selection rules than a single label including revisions modified in an activity or contained in a configuration.

11. Relative URLs appearing in versioned documents (e.g., HTML and XML) which are being edited and/or browsed by a versioning-aware client should work correctly.

    Web resources and client applications often refer to other

resources with relative URLs, an incompletely specified URL that
is completed by pre-pending some known context that would not
contain a revision or workspace name. When used with versioned
resources, these relative URLs may be relative to a versioned
resource or a particular revision. In this case, the context must

include sufficient information for the relative URL to be resolved to a specific revision.

12. If the DAV server supports searching, it should be possible to narrow the scope of a search to the revisions of a particular versioned resource.

    It is often the case that one needs to find, for example, the first revision at which a particular phrase was introduced, or all the revisions authored by a particular person. Given search capabilities for collections, it would be far more sensible to leverage those capabilities than to define a separate search protocol for revision histories. For example, if the server supports [DASL], then the revision histories could be searched via DASL operations.

13. If the DAV server supports searching, revision IDs and label names should be searchable.

    This would allow client applications to search for resources that have a particular revision name. This goal does not specify that any particular search mechanism is implied or needed. It only indicates that labels should be available properties that a search mechanism could access.

14. The CM protocol must be an optional extension to the base versioning protocol.

    It is expected that servers will want to support versioning without supporting configuration management. This goal provides the required flexibility.

15. It must be possible to determine what properties of a checked in revision may change without creating a new revision. Properties of a checked in revision that cannot change are called immutable properties.

    It is anticipated that some properties may be calculated in such a way that their values may change even on a revision that is checked in. Other properties may change without having any effect on the resource itself e.g., review status, approved, etc. This results from the fact that properties may be meta-data about a resource that is actually not describing the state of the resource itself. A client must be able to discover which properties might change in order to maintain its state properly.

16. Revisions are either mutable or immutable. Once an immutable revision has been checked in, its contents and immutable properties can never be changed. A mutable revision can be

checked out, updated, and checked back in without creating a new revision. It must be possible to determine if a revision is mutable or immutable, but the mutability of a revision cannot be changed once it has been checked in.

The concept of mutable revisions is included to support typical document management systems that want to track version histories while allowing more flexible, less formal versioning semantics. Mutable revisions will have some restrictions due to the fact that because the revision may change, certain configuration management semantics cannot be maintained. For example, a mutable revision cannot be a member of a configuration because the configuration would not represent a persistent set of revisions.

17. Each revision may have properties whose values may be changed without creating a new revision. The list of these properties must be discoverable.

    It is expected that certain live properties whose values are calculated by the server may depend on information that is not captured in the persistent state of an immutable revision. The values of these properties may change from time to time without requiring a new revision of the versioned resource. There may also be some DAV properties used to support versioning and configuration management that may change without requiring a new revision.

18. Revisions and versioned resources can be deleted. Generally this is a high-privilege operation. Deleting a revision must update its predecessors' successors.

    This goal is included to support generally necessary maintenance operations on versioning repositories. It is sometimes the case that successors of a revision beyond some point are no longer required and can be removed from the repository to reclaim space. It may also be the case that a versioned resource is no longer used and can be safely deleted. This goal does not intend to express any policy for when or under what circumstance revisions can be deleted. It only provides a mechanism to support particular client or server policies.

19. Once a revision has been deleted, its ID cannot be reused within the same versioned resource.

    In many cases, it is necessary to be able to guarantee (as far as possible) that one can retrieve the exact state of a resource at a particular point in history, and/or all the states which the resource has ever taken on. For example, if a company is sued for violating a warranty that the plaintiff read on the company's Web site, it might be useful to be able to prove that the warranty never contained the provision that the plaintiff says it did. Conversely, it may be useful for the plaintiff to be able to prove that it did. A revision history where all revisions were immutable would provide this sort of ability.

Of course, DAV cannot preclude the possibility of an out-of-band method to change or delete a revision; an implementation may provide an administrative interface to do it. But such access would at least be limited to trusted administrators.

It is possible that a versioned resource contained in a configuration is deleted, and a new, unrelated versioned resource is created using the same URL, and having the same revision id. The configuration may incorrectly include this revision. Requiring revision Ids to be UUIDs would resolve this issue.

20. A configuration can only contain immutable revisions.

    This requirement is included in order to retain the usual semantics of configurations, and to ensure that a configuration can always be recreated. The implication is that unversioned resources, working revisions, and mutable revisions cannot be members of a configuration.

21. It must be possible to query a revision history to learn the predecessors and successors of a particular revision, activity names, the initial and latest revisions, etc.

    If a client wishes to present a user interface for browsing the revisions of a particular versioned resource, it must be able to read the relationships represented within the version history.

22. It should be possible to obtain the entire revision history of a versioned resource in one operation.

    A client wishing to display a map of the revision history should not have to make queries on each individual revision within the revision history. It should be able to obtain all the information at once, for efficiency's sake.

23. The protocol support for parallel development through activities must be an optional capability.

    Activities support controlled parallel development on the same resource, but results in the need to merge multiple changes at some later time. This introduces work and the potential for errors that some servers may want to avoid by requiring updates to be serialized.

24. The protocol must support the following operations:

    1. Creating and accessing revisions:

        . Create a versioned resource from an unversioned resource and set its initial revision to the contents of the unversioned resource. This does not imply that unversioned resources are required. A server could create all resources as versioned resources.

- Obtain the URI of, or access a revision or a versioned resource given the URL for the versioned resource and either a revision name, or a workspace

- Check out a revision in an activity and create a working resource

- Check in a working resource and create either a new revision or update the existing revision in place creating a mutable revision

- Cancel a checkout (delete a working resource)

- Describe a revision with human-readable comments

- See if a resource is versioned

- Get the versioning options for a resource

2. Labels:

   - Apply a label to a particular revision

   - Change the revision to which a label refers

   - Get all the revision names on a particular revision

   - Get the revision history of a resource

3. Activities:

   - Create and name an activity

   - Checkout a revision in an activity

   - Merge an activity into a workspace

   - Generate and maintain the conflict report for a merge

   - Get a list of the resources modified in an activity

   - Apply a label operation to all resources modified in an activity

4. Configurations:

   - Create a configuration

   - Add/remove revisions from a configuration

   - Access a revision given a configuration name that contains it by using a configuration in a version selection rule for a workspace

- Delete a configuration.

- Determine the differences between two configurations by listing the activities in one and not the other.

Some of these operations come from [WEBDAV-GOALS], section 5.9.1.2. Not all of the operations in that section are replicated here; some of them (e.g., locking) fall out naturally from the fact that a revision is a resource.

The protocol must find some balance between allowing versioning servers to adopt whatever policies they wish with regard to these operations and enforcing enough uniformity to keep client implementations simple and interoperable.

25. For each operation that the protocol defines, the protocol must define that operation's interaction with all existing [WebDAV] methods on all existing WebDAV resources.

    This goal applies to all HTTP extensions, not just versioning. However, versioning, parallel development, and configuration management are sufficiently complex and have a broad enough effect on other methods to call out this goal specifically.

26. The protocol should clearly identify the policies that it dictates and the policies that are left up to versioning system implementers or administrators. A client must be able to discover what policies the server supports.

    Many writers have discussed the notion of versioning styles (referred to here as versioning policies, to reflect the nature of client/server interaction) as one way to think about the different policies that versioning systems implement. Versioning policies include decisions on the shape of version histories (linear or branched), the granularity of change tracking, locking requirements made by a server, naming conventions for activities and labels, etc.

27. A client must be able to determine whether a resource is a versioned resource, or whether a resource is itself revision of a versioned resource.

    A resource may be a simple, non-versioned resource, a versioned resource, an immutable revision, a mutable revision, or a working resource. A client needs to be able to tell which sort of resource it is accessing..

28. A client must be able to access a versioned resource with a simple URL and get some well-defined default revision.

    The server should return a default revision of a resource for where no specific revision information is provided. This is one of the simplest ways to guarantee non-versioning client compatibility. This does not rule out the possibility of a server returning an error when no sensible default exists.

It may also be desirable to be able to refer to other special revisions of a versioned resource. For example, there may be a current revision for editing that is different from the default revision. For a graph with several branches, it may be useful to be able to request the tip revision of any branch.

The association of a workspace with a particular user agent for the purposes of applying version selection rules is the responsibility of the client application. The server does not necessarily maintain this association.

29. It must be possible, given a reference to a revision of a versioned resource, to find out which versioned resource that revision belongs to.

    This makes it possible to understand the versioning context of the revision. It makes it possible to retrieve a revision history for the versioned resource to which it belongs, and to browse the revision history. It also supports some comparison operations: It makes it possible to determine whether two references designate revisions of the same versioned resource.

30. Versioning functionality may be partitioned into levels. The lowest level must provide simple versioning of resources and support for labels, checkin, and checkout. Other functions should be as orthogonal as possible so that servers have additional flexibility in choosing features to implement. Functionality at lower levels must be a consistent subset of the functionality at higher levels and not introduce special cases, incompatible, or redundant functions.

    Servers must provide all the functions defined for a given level in order to claim and advertise conformance to that level. A server may choose to implement additional features from higher levels to support particular business and/or client requirements. The OPTIONS method indicates exactly what features are supported while the DAV header indicates the supported level clients can rely on.

    At a minimum, the following actions are actions are available at the basic level:

    . Checkout a revision and receive a way to update the working copy

    . Checkin a working copy to create a new revision

    . Cancel an active checkout

    . Optional for server to support multiple checkouts on the same resource

. Labeling of revisions to identify them

. Access to the linear checkin history of a versioned resource

31. It must be possible to lock an activity so that no one can make further changes in that activity.

32. It must be possible to indicate that a particular resource does not allow parallel development. That is, the resource can effectively only be checked out in one activity.

33. The protocol should be defined in such a way as to minimize the adoption barriers for clients and existing repository managers. This includes integration with legacy data in repository managers supporting the WebDAV protocol.

34. The server must not require client applications to retain state in order to support versioning semantics. That is, a user must be able to begin using versioning with one client, and continue using versioning on some other client at some other time.

35. It must be possible to discover what resources have changed in a workspace from a given point.

36. Versioned resources, revisions, and activities must have an associated URN that is globally unique.

37. Servers may choose to support only mutable revisions, only immutable revisions, or both. Clients must be able to discover the support provided by the server.

38. Activities should be able to be dependent (conceptually include) other activities.

39. Enumeration of versioning resource types should be fast/easy.

1.4 Rationale

Versioning in the context of the worldwide web offers a variety of benefits:

1. It provides infrastructure for efficient and controlled management of large evolving web sites. Modern configuration management systems are built on some form of repository that can track the revision history of individual resources, and provide the higher-level tools to manage those saved versions. Basic versioning capabilities are required to support such systems.

2. It allows parallel development and update of single resources. Since versioning systems register change by creating new objects, they enable simultaneous write access by allowing the creation of

variant versions. Many also provide merge support to ease the reverse operation.

3. It provides a framework for coordinating changes to resources. While specifics vary, most systems provide some method of

controlling or tracking access to enable collaborative resource development.

4. It allows browsing through past and alternative versions of a resource. Frequently the modification and authorship history of a resource is critical information in itself.

5. It provides stable names that can support externally stored links for annotation and link-server support. Both annotation and link servers frequently need to store stable references to portions of resources that are not under their direct control. By providing stable states of resources, version control systems allow not only stable pointers into those resources, but also well defined methods to determine the relationships of those states of a resource.

6. It allows explicit semantic representation of single resources with multiple states. A versioning system directly represents the fact that a resource has an explicit history and a persistent identity across the various states it has had during the course of that history.

1.5 Non-goals

These non-goals enumerate functionality that the working group has explicitly agreed to exclude from this document. They are documented here for explanatory purposes.

1. Revisions in multiple revision histories (see [WEBDAV-GOALS], sections 5.9.1.3 and 5.9.2.5). This capability was felt to be too rarely useful.

2. Federated revision histories (that is, revision histories which are not stored on a single server). This capability would introduce great difficulties. A server implementer who needs it can use out-of-band server-to-server communication. But, this communication is arguably out of the scope of WebDAV, which is a client-to-server protocol. However, the protocol shouldn't do anything to preclude federated version histories at a later date.

3. Client-proposed version identifiers (see [WEBDAV-GOALS], section 5.9.2.8). Labels do the job better.

4. Change management or change control operations. It is envisioned that policies for change management and the mechanisms to implement them will be quite variable for the number and types of users authoring content for the web.

Therefore it is important to provide core capabilities for versioning, parallel development, and configuration management without hindering the policies client applications may choose to present to their users. It is intended that WebDAV versioning will provide these core capabilities, and that a variety of change management policies could be implemented on these core capabilities by client applications.

5. Server-to-server communication (e.g., replication) is not required.

1.6 Security Considerations

To be written. It is likely that implementing features to meet the goals described here will present few or no new security risks beyond those of base DAV. One possible exception is that it may become more difficult to hide the contents of a resource when there may exist other versions with different access control lists.

1.7 References

[WEBDAV]Y.Y. Goland, E.J. Whitehead, Jr., A. Faizi, S.R. Carter, D. Jensen, "Extensions for Distributed Authoring on the World Wide Web -- WEBDAV", Internet-Draft draft-ietf-webdav-protocol-10. Nov., 1998
[WEBDAV-GOALS] J. Slein, F. Vitali, J. Whitehead, D. Durand, "Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web", RFC-2291. February 1998.
[WEBDAV-ACP] J. Slein, J. Davis, A. Babich, J. Whitehead, "WebDAV Advanced Collections Protocol", Internet-Draft draft-ietf-webdav-collection-protocol-02.txt. Nov., 1998.
[DASL] S. Reddy, D. Jensen, S. Reddy, R. Henderson, J. Davis, A. Babich, "DAV Searching & Locating", Internet-Draft draft-reddy-dasl-protocol-04.txt. Nov., 1998.
[CVS] http://www.cyclic.com/cyclic-pages/books.html
[BONSAI] Mozilla.org, http://www.mozilla.org/bonsai.html

1.8 Open Issues

. The current write up of configurations may need to change as we define what a "configuration" is.