DeltaV: Adding Versioning to the Web

WWW10 Tutorial Notes, By Jim Whitehead

Documents change over time. Whether they are word processing documents, spreadsheets, Web pages, or source code, the process of creating the contents of these documents involves change over time. It is frequently useful to track changes to a document over time, as this permits tracking who made a specific change, backing-out undesirable changes, recording why a specific change was made, and the ability to know what the document contents were at a specific point in time. Change tracking capabilities gives greater *control over change*, since changes are now explicitly recorded, and an *archive of important document versions.*

DeltaV is a network protocol that provides facilities for remote versioning and configuration management of documents stored on a Web server. The DeltaV protocol can be used to support the following scenarios:

- A task force of people from geographically dispersed business units need to develop a report together. Throughout the report-writing process, this group needs to solicit feedback, and would like to keep a permanent copy of the exact report version they are having other people review (see Figure 1).
- A cross-company collaboration project involves people from multiple companies, in several different countries. The project has a Web site, which is developed using a WebDAV-aware Web site authoring tool, such as Go Live, or Dreamweaver. Since any project member can edit the Web site contents, the team wants to keep track of all changes, so there is a record of who changed which page, and if a mistake is made the change can be undone.
- An open source project, comprised of team members from around the globe, is collaboratively developing a software application. They need to record all source code changes, as well as create stable baselines of their source code corresponding to public releases of the software. Furthermore, developers need to edit and compile source code on their local machine.

The Web Versioning and Configuration Management (DeltaV) protocol has been developed as an open, standards-based infrastructure that supports these collaborative development scenarios.

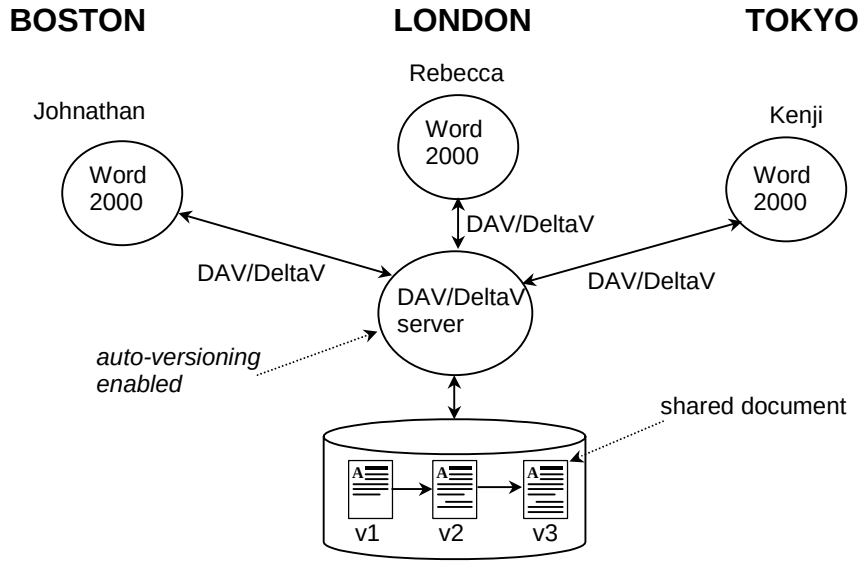

Figure 1 - Three collaborators, located at three different sites, are jointly authoring a document using the WebDAV capabilities of Microsoft Word 2000. The shared document is stored on a DeltaV server, with auto-versioning enabled, and hence the document is automatically versioned as the collaborators work.

The DeltaV protocol is an extension to the WebDAV (Web Distributed Authoring and Versioning) protocol, which itself extends the Hypertext Transfer Protocol (HTTP), the core network protocol that carries Web traffic between a Web server and a Web browser. Despite its name, the WebDAV protocol only provides facilities for remote collaborative authoring of documents, and does not provide any versioning capabilities. Initially the WebDAV working group had wanted to define a protocol for remote versioning, but took much longer than expected just to finish the base remote authoring protocol. As a result, the WebDAV working group postponed work on versioning features. The DeltaV working group picked up where WebDAV left off, taking on the goal of adding versioning to the Web, as well as the more ambitious goal of remote configuration management. The core defining document for the DeltaV protocol is being developed by the DeltaV Working Group of the Internet Engineering Task Force (IETF). Despite being a Web-related standard, the World Wide Web Consortium (W3C) is not actively involved in the DeltaV specification effort.

HTTP protocol operations are called *methods*, and WebDAV adds seven new methods to the set of methods defined by HTTP/1.1 (GET, HEAD, POST, OPTIONS, PUT, DELETE, TRACE). The WebDAV methods provide overwrite protection (LOCK, UNLOCK), metadata management (PROPFIND, PROPPATCH), and namespace management (COPY, MOVE, MKCOL). Just as the user of a Web browser is largely unaware of the HTTP network traffic that request and download Web pages, so too a user of a WebDAV-enabled authoring tool is largely unaware of the use of the WebDAV protocol. The WebDAV protocol is designed to be integrated into existing authoring tools, adding Web-based remote authoring capabilities to the tools users already know how to use. To date, this has been a successful strategy, with WebDAV support in document authoring tools such as Word 2000, PowerPoint 2000, and Excel 2000 (via the "Web Folders" feature) as well as in Acrobat 5 and Photoshop 6, and in Web authoring tools such as Go Live 5, and Dreamweaver 4. You may already own a WebDAV-capable application!

To the base provided by HTTP and WebDAV, DeltaV adds 11 additional methods. Versioning capability is provided by the methods VERSION-CONTROL, CHECKIN, CHECKOUT, UNCHECKOUT, and REPORT. An unversioned resource is put under version control with VERSION-CONTROL. While under version control, a typical editing process begins with CHECKOUT, involves one or more writes (PUTs) to the resource, and ends with a CHECKIN. An editing session can be aborted using UNCHECKOUT. The version history of a resource can be retrieved using REPORT. Unique human-readable names can be associated with specific versions using LABEL. The default visible revision can be set using UPDATE. Two separate branches in a version history can be brought together using MERGE. An *activity* represents logical changes than span multiple revisions; MKACTIVITY creates new activities. A *workspace* allows multiple collaborators to work in isolation on a set of resources; MKWORKSPACE creates new workspaces. A consistent snapshot of a set of resources is a *baseline*, and is useful for recording the state of a software system before major ship dates. The BASELINE-CONTROL method is used to place both versioned and unversioned collections under baseline control, thus creating a version-controlled *configuration*. Checking-out then checking-in a version-controlled configuration creates a new baseline (i.e., a new version of the version-controlled configuration). Figure 2 summarizes the methods defined by HTTP, WebDAV, and DeltaV.

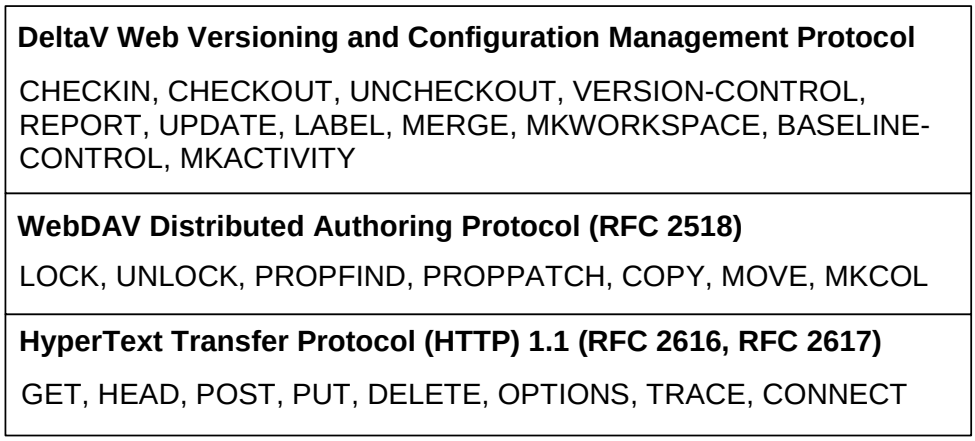

Figure 2 - Methods defined by HTTP 1.1, WebDAV Distributed Authoring Protocol, and the DeltaV Web Versioning and Configuration Management Protocol.

The defining document for the DeltaV protocol is Internet Draft *draft-ietf-deltav-versioning-15*. Internet Drafts are working documents of the IETF, and have not yet been approved by the Internet Engineering Steering Group as a Proposed Standard. That said, the DeltaV protocol has passed through several multi-week working group last call for comments periods, and is expected to be quite stable going forward. Approval as a Proposed Standard is expected by late summer 2001. The current specification is the basis for at least one implementation effort, the Subversion project, an open-source replacement for CVS.

Versioning Data Model and Terminology

Within the HTTP/DAV/DeltaV family of specifications, a document on a Web server is known as a *resource*. Like objects in object-oriented languages, resources have state, and operations on that state. The state of a WebDAV resource comes in two parts, a *body* that contains the primary content (such as the text of a document, or the bitmap data for an image), and *properties*, name/value pairs that provide metadata about the resource. Properties come in two types: *live* properties whose value is computed and controlled by the server, and *dead* properties whose value is controlled by the client, and stored by the server. The operations on resources are termed *methods*, described previously.

DeltaV versioning terminology includes the following terms:

Version Control, Checked-In, Checked-Out

> *Version control* is a set of constraints on how a resource can be updated. A resource under version control is either in a *checked-in* or *checked-out* state, and the version control constraints apply only while the resource is in the checked-in state.

Versionable Resource

> A *versionable resource* is a resource that can be put under version control.

Version-Controlled Resource

> When a versionable resource is put under version control, it becomes a *version-controlled resource*. A version-controlled resource can be *checked out* to allow modification of its content or dead properties by standard HTTP and WebDAV methods.

Version Resource

> A *version resource*, or simply *version*, is a resource that contains a copy of a particular state (content and dead properties) of a version-controlled resource. A version is created by *checking in* a checked-out resource. The server allocates a distinct new URL for each new version, and this URL will never be used to identify any resource other than that version. The content and dead properties of a version never change. Version resources are sometimes termed *revisions*.

Version History Resource

A *version history resource*, or simply *version history*, is a resource that contains all the versions of a particular version-controlled resource.

Version Name

A *version name* is a string chosen by the server to distinguish one version of a version history from the other versions of that version history (e.g., "1.0", "1.2.1"). Versions from different version histories may have the same version name. Version names are sometimes called *version identifiers*, or *version numbers*.

Label

A *label* is a name that can be used to select a version from a version history. A label can be assigned by either a client or the server. The same label can be used in different version histories.

Fork, Merge

When a second successor is added to a version, this creates a *fork* in the version history. When a version is created with multiple predecessors, this creates a *merge* in the version history. A server may restrict the version history to be linear (with no forks or merges), but an interoperable versioning client should be prepared to deal with both forks and merges in the version history.

Figure 3 below illustrates several of the previous definitions.

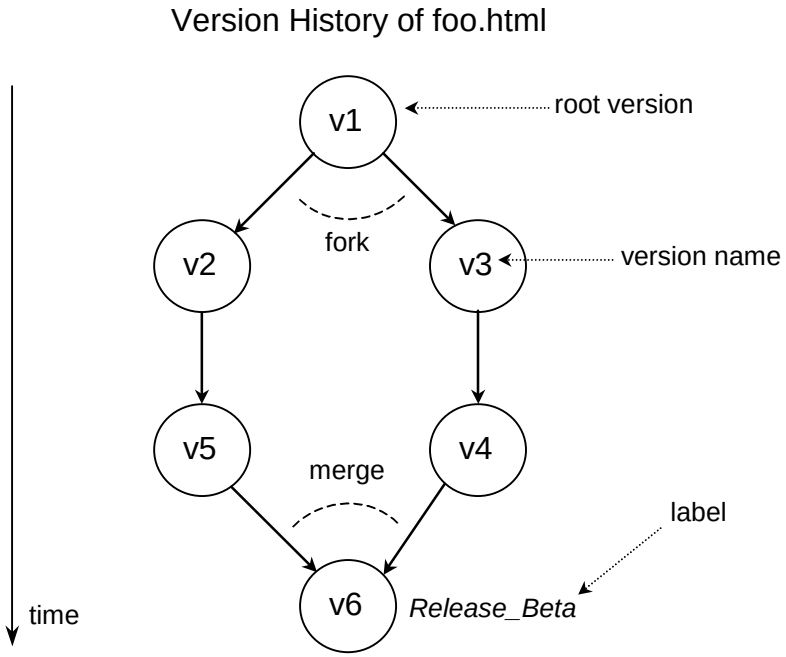

Figure 3 - A sample version history of a Web page.

In order to track the history of the content and dead properties of a versionable resource, an author can put the resource under version control with a VERSION-CONTROL request. A VERSION-CONTROL request performs three distinct operations:

1) It creates a new *version history resource*. By default, a version history resource is not assigned a URL, and hence is not visible in the http scheme URL space. However, when the optional version-history feature is supported, this changes, and each version history resource is assigned a new distinct and unique server-defined URL.

2) It creates a new *version resource* and adds it to the new version history resource. The body and dead properties of the new version resource are a copy of those of the versionable resource. The server assigns the new version resource a new distinct and unique URL.

3) It converts the versionable resource into a *version-controlled resource*. The version-controlled resource continues to be identified by the same URL that identified it as a versionable resource. As part of this conversion, it adds a *DAV:checked-in* property, whose value contains the URL of the new version resource.

This versioning data model is different from versioning tools such as RCS and SCCS. Taking RCS as an example, placing a file "foo.html" under version control causes foo.html to be made read-only, and an archive file RCS/foo.html,v is created. The archive file provides storage for the version history and for each version of foo.html. Thus, the archive file plays the role of the version history resource and the set of version resources. The read-only foo.html is similar in function to the version-controlled resource, acting as the point in the namespace where versioning commands are directed (i.e., typically a user explicitly names foo.html when they perform a checkout, instead of naming RCS/foo.html,v). Figure 4 below summarizes the triad of resources used to represent a resource under version control, and its version history.

The value of the separation between version-controlled resource, version history resource, and version resources lies in the ability to have multiple version-controlled resources for a given version history and its versions. The version-controlled resource can be viewed as a handle for a version history which can be placed in the authoring namespace of a server. The version-controlled resource is the primary resource to which versioning commands such as CHECKOUT and CHECKIN are addressed. The benefit of having multiple version-controlled resources per version history comes when workspaces are used. A workspace is a copy of a set of resources in which a collaborator can work in isolation from other collaborators. A workspace is made by creating a separate version-controlled resource for each workspace. This allows multiple collaborators to work on a given version history in relative isolation; their interactions will come in the form of branches in the version history. If the version-controlled resource were merged with the version history, it would be more difficult to create workspaces, since the handle for the version history, and the version history itself would be combined.

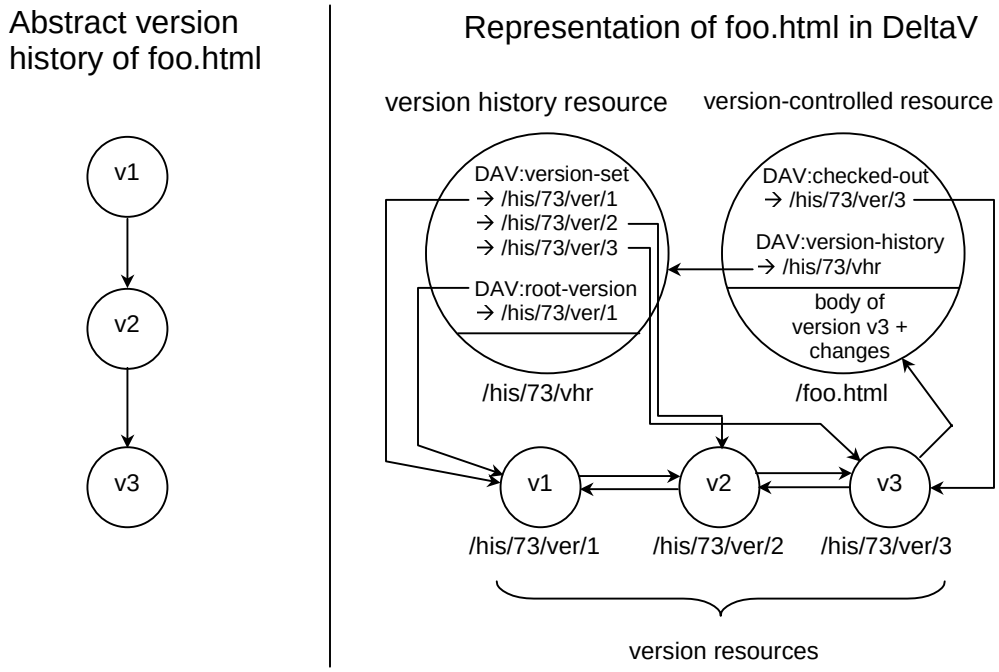

Figure 4 – Representation of a linear version history in the DeltaV data model. Since it is checked-out, and hence writeable, the body of the version-controlled resource is the same as the body of version v3, plus any changes that have been made since checkout. Information above the line in the version history resource and the version-controlled resource are properties. Each arrow-terminated line represents a URL acting as a pointer. The predecessor relationships between version resources are stored in the DAV:predecessor-set property (shown in the figure only as arrow-terminated lines) on each version; similarly, successor relationships are stored in the DAV:successor-set property (also only shown as arrow-terminated lines). Version 3 additionally has the URL of the version-controlled resource in its DAV:checkout-set property

(shown as an arrow-terminated line). The version name (“v1”, “v2”, “v3”) is stored in the DAV:version-name property on each version.

Versioning Operations

Checkout, Checkin, and Uncheckout

The CHECKOUT method is applied to a version-controlled resource to allow modifications to its content and dead properties. When a version is already checked out, or already has a successor, it is possible that a branch could be made in the version history. Creation of a new branch upon checkout is controlled by the *DAV:checkout-fork* property, which is stored on version resources (not the version-controlled resource). Possible fork-control options include *ok*, a fork can be created, *discouraged*, the client explicitly needs to indicate that fork creation is fine, and *forbidden*, a fork cannot be created. A similar property, called *DAV:checkin-fork* controls creation of branches during CHECKIN.

While in the checked-out state, authoring clients can modify the content and dead properties of the version-controlled resource using PUT and PROPPATCH. Once they are finished editing, the state of the resource can be frozen as a new version using the CHECKIN method. If the editing session needs to be aborted, and reverted back to the state it held just prior to the checkout, the UNCHECKOUT method can be invoked.

A comment describing the purpose of a change can be written into the *DAV:comment* property of a version.

Autoversioning

By the time there are DeltaV-capable servers, there will be a significant number of WebDAV clients. These clients will not have any notion of versioning (i.e., they will not understand the CHECKOUT and CHECKIN methods), yet it is desirable to provide versioning support for them. Ideally, as a document is edited using a WebDAV-capable client, the DeltaV server will automatically version its contents. DeltaV provides this capability, using a feature known as *auto-versioning.*

There are two styles of auto-versioning, depending on whether a new version is created every time the resource is updated, or only when the resource is unlocked. In the first case, a PUT/PROPPATCH (i.e., a state-modifying request) is preceded by a CHECKOUT, and is followed by a CHECKIN. That is, a PUT/PROPPATCH is expanded to CHECKOUT→PUT/PROPPATCH→CHECKIN. In the second auto-versioning style, a LOCK request also results in a CHECKOUT, and an UNLOCK additionally results in a CHECKIN. This works well for authoring clients that using locking, since they typically take out a lock at the start of an editing session, and remove the lock at the end, and hence auto-versioning based on locks causes the authoring session to automatically be bracketed by CHECKOUT and CHECKIN operations.

Labels

Labels are human-readable strings that can be attached to a particular version. Labels are guaranteed to be unique within a version history, but can be reused across version histories, and thus a label like “release_beta” could be used to identify a single revision in multiple version histories. Labels are often used as a simple form of configuration management; a configuration is the set of all versions with a given label. The drawback to this scheme is that the mapping of label to version can change over time, and hence there is no guarantee a configuration will remain the same. Labels are also used for selecting a specific version from a version history; the *Label* header can be used with a GET and PROPFIND on a version-controlled resource to select a specific version. The LABEL method is submitted to a version to add a new label, move a label to that version, or delete a label.

Update

Typically, the body and dead properties of a checked-in version-controlled resource are the same as those of the last checked-in version. For a linear version history, a checked-in version controlled resource typically follows the tip of the history. The UPDATE method modifies the content and dead properties of a checked-in version-controlled resource to be those of a specified version from the version history of that version-controlled resource. This allows some version other than the one last checked-in to be reflected by the version-controlled resource.

Merge

When a version history has one or more branches that represent the output of collaborators working in parallel (as opposed to representing variants), it is useful to merge together the branches. MERGE performs this operation. If the server understands how to merge the two selected revisions together (most likely this will be the case for text, as in program source code), it will do so. Otherwise the client has the responsibility to combine together the contents and dead properties of the two revisions. This can be performed by displaying a graphical merge tool to the user. One option of MERGE (DAV:no-auto-merge) forces the server not to automatically merge the contents.

Version Tree Report

Versioning applications often provide in their user interface a depiction of a version history. There are two ways a DeltaV client can gather the information needed to create such a visualization. The first is to retrieve the contents of the *DAV:version-set* property of the version history, and then retrieve the *DAV:successor-set* and *DAV:version-name* properties for each version in the history. For an *N* version history, this would require *N+1* PROPFIND requests. To provide a more efficient mechanism to retrieve this same information, the report method was created. There are many possible reports that can be generated by a DeltaV server. The version tree report allows a client to request a specific set of properties from all members of a version history. By requesting the *DAV:successor-set*, and *DAV:version-name* properties in the REPORT request, a client can retrieve all of the information needed to create a version history visualization in one network round trip.

Activities

An *activity* is a non-versionable resource that selects a set of versions that are on a single *line of descent*, where a line of descent is a sequence of versions connected by successor relationships. If an activity selects versions from multiple version histories, the versions selected in each version history must be on a single line of descent.

A common problem that motivates the use of activities is that it is often desirable to perform several different logical changes in a single workspace, and then selectively merge a subset of those logical changes to other workspaces. An activity can be used to represent a single logical change, where an activity tracks all the resources that were modified to effect that single logical change. When a version-controlled resource is checked out, the author specifies which activity should be associated with a new version that will be created when that version-controlled resource is checked in. It is then possible to select a particular logical change for merging into another workspace, by specifying the appropriate activity in a MERGE request.

Another common problem is that although a version-controlled resource may need to have multiple lines of descent, all work done by members of a given team must be on a single line of descent (to avoid merging between team members). An activity resource provides the mechanism for addressing this problem. When a version-controlled resource is checked out, a client can request that an existing activity be used or that a new activity be created. Activity semantics then ensure that all versions in a given version history that are associated with an activity are on a single line of descent. If all members of a team share a common activity (or sub-activities of a common activity), then all changes made by members of that team will be on a single line of descent.

Activities appear under a variety of names in existing versioning systems. When an activity is used to capture a logical change, it is commonly called a *change set*. When an activity is used to capture a line of descent, it is commonly called a *branch*. When a system supports both branches and change sets, it is often useful to require that a particular change set occur on a particular branch. This relationship can be captured by making the change set activity be a *subactivity* of the branch activity.

The MKACTIVITY method creates new activities. This activity is then submitted as part of a CHECKOUT operation, which then causes the server to record the URL of the checked-out resource in the *DAV:activity-checkout-set* property. When the checked-out resource is checked-in, the URL of the new version is recorded in the *DAV:activity-version-set* property, and the checked-out resource is removed from *DAV:activity-checkout-set*.

Workspaces

A workspace is a location where a person can work in isolation from the ongoing changes made by all other collaborators working on the same set of resources. There are two broad classes of workspace, client-side and server-side.

A *client-side workspace* is one in which copies of all resources in a project have been replicated to the local disk of the client, and all editing work takes place on the local replica (this is how CVS works). Once editing has been finished, the contents of a local file are written to the server (using PUT), and are then checked in. Client-side workspaces have the advantage of good support for disconnected operation, and the ability to work with all existing file-oriented tools (this advantage is also shared by server-side workspaces that employ local caching). Data-intensive activities, like code compilation, or static code analysis, typically work faster on locally cached data. Client-side workspaces have their drawbacks, though. They do not allow a user to access the workspace from clients in different physical locations, such as from another office, from home, or while traveling. Client-side workspaces do not isolate clients from a logical change that involves renaming shared resources; all clients use a common set of shared version-controlled resources and every client sees the result of a MOVE as soon as it occurs.

A *server-side workspace* is one in which there are multiple locations in the server URL namespace from which to access the set of project resources. Typically there is a separate location for each collaborator, so, for example, Lisa might have her workspace at `/users/people/lisa/projectX`, and Chuck might have his workspace at `/users/people/chuck/projectX`. Multiple version-controlled resources per version history (one per collaborator) allow parallel work on the same version history from multiple workspaces. Like client-side workspaces, server-side workspaces also permit local replication of data, and implementations will typically cache at least a portion of each version history. Server-side workspaces can be accessed from multiple locations, and permit logical operations such as a MOVE to be kept isolated until it is shared with other collaborators. The drawback of server-side workspaces is the additional demands they place on the server namespace, requiring a separate portion of the namespace for each collaborator.

Client-Side Workspaces and Working Resources

Since the client maintains a client-side workspace, very little server state maintained. Server support for client-side workspaces comes in the form of the *working resource* feature. A working resource is created upon checkout, and is a location on the server where the client can write the contents of the checked-out resource once it is ready to be checked-in. So, for example, a client using client-side workspaces would first replicate the contents of resource `/foo.html` by performing a GET `/foo.html`, and writing the GET response to a local disk under the name `/{local workspace name}/foo.html`. The client would next perform a CHECKOUT on the version it just retrieved using GET, thus causing the creation of a working resource, whose URL is returned in the *Location* header. The client stores the working resource URL locally. At this point, the client can, if it wishes, completely disconnect from the network while editing takes place. Once editing is done, the client writes the new value of `/{local workspace name}/foo.html` to the working resource on the server using PUT, and follows this with a CHECKIN of the working resource, causing the server to record the contents of the working resource as a new version resource.

Server-Side Workspaces

For server-side workspaces, the server explicitly records the membership of a workspace. A new workspace is created using the MKWORKSPACE method. Workspaces can contain versioned and unversioned resources. A workspace is created by building up its contents, resource by resource, either by adding new unverisoned resources using PUT, or adding existing versioned-controlled resources using VERSION-CONTROL.

For example, consider the following workspace containing three version-controlled resources:

```
/projectX/makefile
/projectX/main.c
/projectX/defs.h
```

Geoff and Chris want to collaborate together on this project, each working in a separate server-side workspace. First, the two workspaces are created:

MKWORKSPACE `/users/geoff/projectX/`
MKWORKSPACE `/users/chris/projectX/`

Next, version-control is used to add each of the version-controlled resources to the two workspaces. First Geoff's workspace is populated:

VERSION-CONTROL `/users/geoff/projectX/makefile` from `/projectX/makefile`
VERSION-CONTROL `/users/geoff/projectX/main.c` from `/projectX/main.c`
VERSION-CONTROL `/users/geoff/projectX/defs.h` from `/projectX/defs.h`

Then Chris' workspace is populated:

VERSION-CONTROL `/users/chris/projectX/makefile` from `/projectX/makefile`
VERSION-CONTROL `/users/chris/projectX/main.c` from `/projectX/main.c`
VERSION-CONTROL `/users/chris/projectX/defs.h` from `/projectX/defs.h`

At the end of this sequence, there are three server-side workspaces (the original workspace, Geoff's workspace, and Chris' workspace), each containing the three project resources. The version histories associated with each of the project files now has three version-controlled resources associated with it, one for each of the two workspaces, and the one it had originally.

Baselines

A *configuration* is a set of resources that consists of a root collection and all members of that root collection that are not members of another configuration. A configuration that contains a large number of resources can consume a large amount of space on a server. This can make it prohibitively expensive to remember the state of an existing configuration by creating a copy of its root collection.

A *baseline* is a special kind of version resource that captures the state of the version-controlled members of a configuration. In particular, it captures the DAV:checked-in version of each version-controlled resource that is a member of the root collection, as well as the DAV:checked-in version of the collection if the collection itself is a version-controlled resource. A *baseline history* is a special kind of version history whose versions are baselines. New baselines are created by checking out and then checking in a special kind of version-controlled resource called a *version-controlled configuration*.

A collection that is under baseline control is called a *baseline-controlled collection*. In order to allow efficient baseline implementation, the state of a baseline of a collection is limited to be a set of versions and their names relative to the collection, and the operations on a baseline are limited to the creation of a baseline from a collection, and restoring or merging the baseline back into a collection. A server can automatically put a collection under baseline control when it is created, or a client can use the BASELINE-CONTROL method to put a specified collection under baseline control.

As a configuration gets large, it is often useful to break it up into a set of smaller configurations that form the logical *components* of that configuration. In order to capture the fact that a baseline of a configuration is logically extended by a component configuration baseline, the component configuration baseline is captured as a *subbaseline* of the baseline.

The root directory of a configuration is unconstrained with respect to its relationship to the root collection of any of its components. In particular, the root directory of a configuration can be an ancestor of a root directory of one of its components (e.g. configuration /sys/x can have a component /sys/x/foo), a descendant (e.g. configuration /sys/y/z can have a component /sys/y), or neither (e.g. configuration /sys/x can have a component /comp/bar).

Further Reading

WebDAV Distributed Authoring Protocol:

- WebDAV Resources
  <http://www.webdav.org/>
  A web site containing a central collection of pages and links to all things WebDAV.
- WebDAV Working Group
  <http://www.ics.uci.edu/pub/ietf/webdav/>
  Contains links to active documents, and a complete list of WebDAV-supporting applications.
- Y. Y. Goland, E. J. Whitehead, Jr., A. Faizi, S. R. Carter, D. C. Jensen, "HTTP Extensions for Distributed Authoring – WebDAV", Microsoft, U.C. Irvine, Netscape, Novell. RFC 2518, February, 1999.
  <http://www.ics.uci.edu/pub/ietf/webdav/protocol/rfc2518.pdf>
  The WebDAV Distributed Authoring Protocol specification.
- E. J. Whitehead, Jr., Y. Y. Goland, "WebDAV: A network protocol for remote collaborative authoring on the Web", *Proc. of the Sixth European Conference on Computer-Supported Cooperative Work*, Sept. 12-16, 1999, Copenhagen, Denmark, pp. 291-310.
  <http://www.ics.uci.edu/~ejw/papers/dav-ecscw.pdf>
  An academic paper giving an overview of the WebDAV Distributed Authoring Protocol.
- Evolving the Web into a Read and Write Medium: An Interview with Jim Whitehead, Chair IETF WebDAV
  <http://msdn.microsoft.com/workshop/standards/webdav.asp>
  A good series of Q&A concerning WebDAV.
- WebDAV Book of Why, Yaron Y. Goland
  <http://www.webdav.org/papers/#misc>
  A collection of perspectives on WebDAV design rationale
- WebDAV Frequently Asked Questions (FAQ)
  <http://www.webdav.org/other/faq.html>
  A slightly dated list of DAV Q&A

Versioning and Configuration Management (Delta-V)

- *Delta-V Working Group web page*
  <http://www.webdav.org/deltav/>
  The home page for the IETF Delta-V Working Group, with links off to the most recent specifications.
- *G. Clemm, J. Amsden, C. Kaler, J. Whitehead, "Versioning Extensions to WebDAV", Internet-Draft, work-in-progress,* draft-ietf-deltav-versioning-15,*April 14, 2001.*
  <http://www.webdav.org/deltav/protocol/draft-ietf-deltav-versioning-15.htm>
  The most recent revision of the versioning and configuration management protocol specification.
- *J. Whitehead, "The Future of Distributed Software Development on the Internet." Web Techniques, Vol. 4, No. 10, October, 1999, pages 57-63.*
  < http://www.webtechniques.com/archives/1999/10/whitehead/>
  An introduction to WebDAV and DeltaV that describes the advantages of DeltaV over CVS for remote collaborative software development.