WS-Workspace: Workspace Versioning for Web Services

Garret Swart

Computer Science Department
University College Cork
Cork, Ireland
g.swart@cs.ucc.ie.

Abstract. When using web services to perform complex data manipulations, users and administrators need control over how their changes are managed and seen by other clients of the service. This includes support for undo of changes, batch publishing of many changes, 'what if' analysis, the collaboration of several people in making and approving a complex change, workspace based access control, and the auditing and tracking of changes. We propose taking the workspace versioning model, used extensively in CAD and CASE products, and using it to augment web services in a backward compatible way based on the WS-Coordination protocol. The resulting protocol, which we call WS-Workspace, facilitates the writing of web services that support applications with undo, collaboration, and auditing.

1 Introduction

The evolution of data access in web services is following a path that traditional web applications have already evolved along. At the dawn of the web, web applications filled their first niche as information navigators, they then adapted to meet the needs of commerce by developing order entry capability, and now, as the web is being used as a delivery vehicle for applications of all types, web applications are struggling to meet the needs of more general online data authoring and manipulation.

Web Services are going through a distributed version of this same evolution. The first applications for web services were oriented towards distributed information access, allowing applications like comparison-shopping, contacting many product sources to find the lowest price; information portals, collecting information from a variety of information sources and presenting them in the same web page (e.g. [13]); and application integration (e.g. [20]), integrating application functionality from various vendors onto a provider's web site. Evolving standards provide support for these uses of web services.

Distributed order-entry web services are in development, allowing the reliable and secure placing and accepting of orders, allowing both horizontally distributed applications, such as booking an itinerary on a variety of carriers and hotels, and vertically distributed applications, such as automatically matching incoming inventory to out

M.E. Orlowska et al. (Eds.): ICSOC 2003, LNCS 2910, pp. 351–366, 2003.

going orders. These applications require coordination between the services to ensure that partial updates are not committed. For example, to make sure that a partial itinerary is not booked or that an order for additional inventory is not confirmed before the order for the finished product is confirmed. Proposals such as WS-Transaction protocol [7] are addressing these issues.

The next stage of web service evolution is to allow for more general online authoring and manipulation of data. Complex data types and web-based maintenance of these values are becoming more common in server-based applications as the complexity and richness of data maintained by web applications is increasing. System architects would like to use web services for applications with complex authoring needs such as:

- Resource scheduling: Defining resources, constraints, resource classes, and resource demands and allowing the authoring and editing of schedules, either automatically or manually.
- Work-flow management: Defining processes and flows and maintaining the state of orders, inventories, claims, or sales calls.
- Catalogue maintenance: Authoring online catalogues of all types, for example a school's class offerings, a mail order firm's product offerings, or a service company's service offerings.
- Business Rules maintenance: Online authoring of rules that determine how a system works, e.g. how it determines pricing, which products to present to the customer, how to deal with delinquent customers, or how the work flow should be routed.
- Online content authoring: From Web Logs to Photo Albums to RFP preparation, users are authoring complex entities constructed from many components.

In each case, the objects being authored are complex, have interactions with other objects and the changes need to be tested by humans before they are published widely. Only the last example is a traditional authoring application, however, taking an authoring point of view on all these applications can make understanding the problem and its solution easier. Note that in each case a particular user update may require changes made to data that is accessed through many different web services and stored on many different underlying data stores.

Web services underlying such applications would benefit from facilities for:

- Batched Publishing: The ability for a caller to specify that the changes being made should be held until the entire batch is completed and then published as a unit for the other callers of the service.
- 'What if' analysis: The authoring of a possible future state so that it can be analyzed to determine whether the effort should be rolled back or continued. A cautious management team may want to see a complete picture of a future state before any steps in that direction are taken.
- Undo: The ability for a caller that has just made a change to issue a subsequent undo request that would undo the last change.
- Collaborative Workspaces: The ability for a caller to make an update in a workspace that may be shared and further modified by a select group of other users.

- Controlled Update: The ability for management to define processes for changing certain data, e.g. all changes must first be tested by QA, approved by marketing and signed off by legal before being published on a public site, and making sure that the web services manipulating the data, enforce those rules.
- Auditing: The ability to see what changes were made and who made them.
- Coordinated Update: The ability to combine changes made through different web services into one perceived global change to the system state.

Most web services, and in fact most web applications, do not offer any of these facilities today. Commonly, each valid update a caller performs using a web service is immediately committed to the database to be seen by all other callers of the web service. Undo is usually nonexistent: once an update is performed, the previous state has been lost. Programmers implement auditing on a piecemeal basis. Sharing work in progress is not a concept in most web services.

In a separate paper

We propose to make the web services world ready for complex object authoring by introducing the concept of workspace versioning to web services and the data that they manipulate. Workspace versioning has been used extensively in the CASE [12] and CAD industry but it is not generally used in server based applications. This extension builds on the WS-Coordination extension [6]. It allows changes made by a web service invocation to create new versions of the updated objects that are part of a particular workspace.

In this paper we are discussing the versioning of the data being manipulated by a web service, not the versioning, compatibility issues or configuration of the web service implementation or its WSDL interface. That is a separate problem that is not addressed in this paper.

Existing work in making versioned updates via the web has been done primarily in the context of content and source management. WebDAV [11, 14] provides a protocol to allow documents to be versioned and published. The Wiki repository [30] allows documents to be added and updated using a web interface. Commercial content management systems like Interwoven [0] and Vignette [29] provide interfaces for updating objects living in a virtual versioned file system.

Traditional CAD and CASE systems are now also allowing access to their systems via the web, very commonly for viewing and sometimes for update.

2 Workspace Versioning Concepts

There is a rich literature and practice in versioning, especially for object and document systems [8, 10, 17, 18, 26]. Here we generalize and formalize the traditional versioning concepts so that they can be applied to arbitrary data objects.

Data Store: A set of data items. This may be a file system, a relational database, an object store or an ERP system. The operations that may be issued on a data store and the data items that it supports depend on the type of the data store and the APIs that it supports. A single web service may use many data stores and a single data store may be used by many web services.

Transactional data store: A data store that supports transactions. To facilitate transactions that span data stores, data stores typically register themselves as resource managers with a transaction manager as they are accessed by a transaction. The data store is then invoked as the transaction manager coordinates a rollback or a commit of the transaction [1, 9, 21, 22].

Data item: A uniquely identified item of data present in a data store. This may be a row in a relational store, an object in an object store, a file in a file store, or an account in an ERP system. A data item may have a complex value with many properties associated with it, including references to other data items. Each data item is assumed to get a unique identity when the data item is first created and for that identity never to change even as changes are made to the value of the item. In this way the item's identity is distinct from a mutable primary key in a relational database or a file name in file system.

Data item version: A value of a data item to be made visible in a particular context. More formally it is a node in an acyclic directed graph associated with the data item, each node in the graph is labeled with a sequence of operations that were used to create that version. The value of each version is the result of applying that sequence of operations. Applying an operation to an existing data item version creates a new version with an arc between the old version and the new one. The original version is called the predecessor and the resulting version is called the successor. The new version is labelled with the new operation appended to its predecessor's label. Since operations may be applied to any version of a data item, a version may have many successors. The version of a data item that results when a **delete** operation is applied has a special value that precludes any additional operations being applied to it. An **insert** operation creates a new data item with a single version node.

Versioned data item: A data item that may have more than one version in its data store. Not all versions of a data item must remain accessible, only those that can be accessed by active workspaces.

Common ancestor: The first common version that may be reached by following back through the predecessors of two versions of the same data item. Since all data items start as a single version, created by the **insert** operation, any two versions of the same data item must have a common ancestor.

Data item merge: An operation on two versions of the same data item resulting in a new data item version created by merging the operations applied to each version since their common ancestor. The resulting data item version has both data item versions as its predecessors. The semantics of the merge operation depends on the data store and the data item; some merges can be done automatically by the data store – when the operations applied to each version since their common ancestor commute – while others require program or even human intervention to resolve. Merges that cannot be resolved automatically are called merge conflicts. The merge operation itself must be commutative and the merge of a version with any of its ancestors or itself, results in the same version. That is, given two versions x and y of the same object with common ancestor z, then

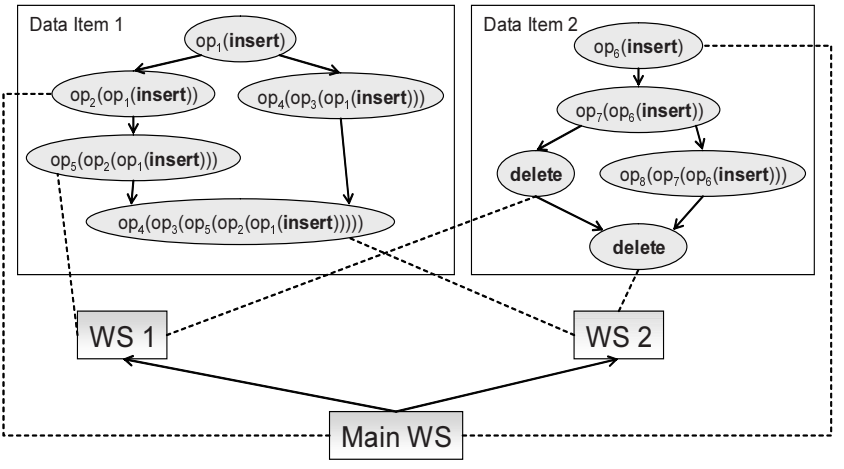

Fig. 1. Illustration of three workspaces and two data items. Each workspace contains a version of each data item. The versions of each data item form a directed graph rooted an initial insert operation. Two merge operations are also illustrated on the left showing how commutative operations are merged, and on the right, how a delete operation might take precedence

$$\textbf{merge}(x,y) = \textbf{merge}(y,x)$$
$$\textbf{merge}(x, z) = x$$
$$\textbf{merge}(y,z) = y$$

Versioned Data Store: A data store that holds versioned data items.

Workspace: A partial function from the set of data items to the corresponding data item versions. The range represents the data items that are available in the workspace. The range defines the values that are operated on by transactions executing in the context of this workspace. For example, above in Fig. 1, the domain of WS1 is the set consisting of data item 1 and data item 2 while the range is the data item version labeled "$op_5(op_2(op_1(insert)))$" associated with data item 1 and the deleted value associated with data item 2.

Update: A set of operations to apply to a workspace atomically. More formally it is a partial function from the set of data items to a sequence of operations that is to be applied to that data item. An update is applied to a workspace by applying the operation sequences to the corresponding data item versions from that workspace, creating new successor versions, and updating the range of the workspace to include those new versions in place of the old versions. If an operation is an **insert**, then a new data item is added to the domain of the workspace. Data store consistency constraints should be checked in each workspace as operations are applied.

If W_i is the version of data item i inside workspace W and **domain**(W) is the set of data items which have versions in W, and U is the update whose value for data item i

is U_i, then the range of the workspace W' that results from applying the update U to W is

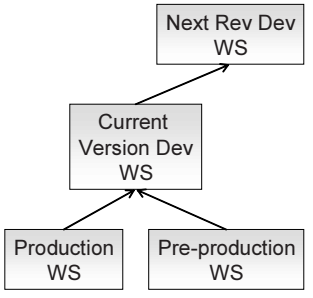

Fig. 2. Workspace Relationships.

$$\bigcup_{i \in \mathbf{domain}(W) \cap \mathbf{domain}(U)} U_i(W_i)$$
$$\cup \bigcup_{i \in \mathbf{domain}(U) - \mathbf{domain}(W)} U_i(\mathbf{insert})$$
$$\cup \bigcup_{i \in \mathbf{domain}(W) - \mathbf{domain}(U)} W_i$$

Workspace Branch: A child workspace created by making a copy of an existing parent workspace. Changes in the child or the parent workspace do not affect each other. This is used to create a context for making changes that are not seen by users of other workspaces. It can also be used to implement what some version management systems call a snapshot or a label. Workspaces are generally named to allow them to be more easily referenced by users.

Workspace Merge: The workspace that results from computing the union of the data item merges of the data item versions from one workspace with the corresponding data item versions from a second workspace and any data item versions from either workspace which do not have corresponding versions in the other. The range of the merge is of workspaces S and T is:

$$\bigcup_{i \in \mathbf{domain}(S) \cap \mathbf{domain}(T)} \mathbf{merge}(S_i, T_i)$$
$$\cup \bigcup_{i \in \mathbf{domain}(S) - \mathbf{domain}(T)} S_i$$
$$\cup \bigcup_{i \in \mathbf{domain}(T) - \mathbf{domain}(S)} T_i$$

The merged workspace typically replaces one of the original workspace when the merge is performed in the context of that workspace. The merged workspace should be verified to satisfy all of the data store's consistency constraints. A workspace merge replacing the parent workspace is typically used as the commit mechanism in a version based long running transaction system. Such a merge would be successful only if there were no conflicts in any of the individual item merges.

Check out: An exclusive lock held by a child workspace on a data item version inside of the parent workspace. A check out lock is acquired to preclude the introduction of any successor version of the version in the parent. The lock is typically released when the child merges into the parent workspace. To reduce the chance of merge conflicts on a change, one should acquire a check out lock for the affected objects in all the workspaces where merges are anticipated. Note that this does not preclude parallel activity when that activity is localized in a portion of the workspace tree unaffected by the check out lock.

Main Workspace: A distinguished primordial workspace that is the parent to all other workspaces. Note that the version of the data items in the main workspace is sometimes called the 'current' version of a data item.

In practice a more complex strategy with several distinguished workspaces is generally used. For example, an enterprise may maintain a production workspace being used by customers, a pre-production workspace getting final approval from management, a development workspace holding the latest integrated version of the current web site, and another development workspace with a partially integrated next release of the web site. This is illustrated in Fig. 2.

3 Workspace Versioning

Orthogonal versioning refers to the process of taking an existing data access API and turning it into a versioned data access API while supporting the original API and defining the default behaviour so that version unaware applications will still behave naturally. This was the approach that was taken when Microsoft Repository was upgraded from the version-less Version 1 to the version and workspace supporting Version 2 [3] and argued for independently in a manuscript by [19]. It was also a guiding principle used more recently in defining the semantics in the Oracle Workspace Manager [23].

We extend the notion of orthogonal versioning to say that versioning should not only be orthogonal to the applications, but should be orthogonal to the web service protocol being extended.

Rather than taking each web service protocol and independently extending it with its own notion of versioning; we use the notion of a coordinator from the WS-Coordination proposal and extend it to manage versioning and workspaces across all web services using a single extension. In the calling language, ideally, the workspace coordinator context is bound to the thread, allowing the workspace to be an implicit parameter to the web services' language level access API.

The advantages of this approach are:

1. Workspaces and undo can span many web services.
2. Having a single versioning approach for all web services reduces the conceptual load on the programmer.
3. A layered coordination protocol means that web service definitions do not need to change extensively when versioning is supported; they just have to refer to the WS-Workspace coordination protocol.

The first advantage is very important as in many applications a single user perceivable change is made up of many updates to many different web services. Just as traditional distributed transactions and WS-Transaction services allow for short lived transactions to extend beyond a single data store, so too long lived transactions and workspaces can be extended to span several data stores. These data sources might include relational databases, object layers on top of relational databases, Enterprise Resource Planning (ERP) systems, and content management systems. An example may include a workspace that contains both changes to a site's content, stored in a

Table 1. Version Aware Applications and Web Services.

	Version Aware Web Service	Unversioned Web Service
Version Aware Application	Application specifies workspace	Special code needed
Version Oblivious Application	Use distinguished workspace	Use current state

content management system, and corresponding changes to a site's database to refer to that content. Having a workspace containing related changes in several data stores that can be merged into a production workspace in an atomic action is a new and powerful capability.

In addition to having version aware and version oblivious applications, an application may access versioned and unversioned web services. Consider all four cases illustrated in Table 1.

A version aware application using a version aware set of web services will specify the workspace to use and issue any needed branch and merge operations it needs. A version oblivious application using a version aware set of data stores will need to execute in the environment of some workspace, the obvious one being the workspace distinguished by the service or configured when the application is deployed. A more complex case is when a version aware application is using an unversioned data store. In this case the application is expecting certain semantics that the data store may not provide. This is analogous to a transactional program using a data store that does not provide transactions, e.g. a traditional file system. It can be done, but it has to be done carefully.

Extending an existing application for versioning can be quite simple. For example, an existing web application might store the logged in user in its session state. A versioned extension of that application might be configured to also store a workspace context in its session state. To minimally Workspace enable the application, only one new UI element may be required: A form to trigger the session's workspace to be merged into the main application workspace – a long running transaction commit.

4 Workspace Versioning Application Protocol

A sample application protocol for workspace versioning is described below. This protocol extends the coordination protocol similar to the way that the WS-Transaction extends the WS-Coordination protocol for the atomic transaction (AT) and business transaction (BT) protocols.

The operations defined below are the basics needed to meet the requirements of the web server applications that are the primary target of this effort. Many more entry

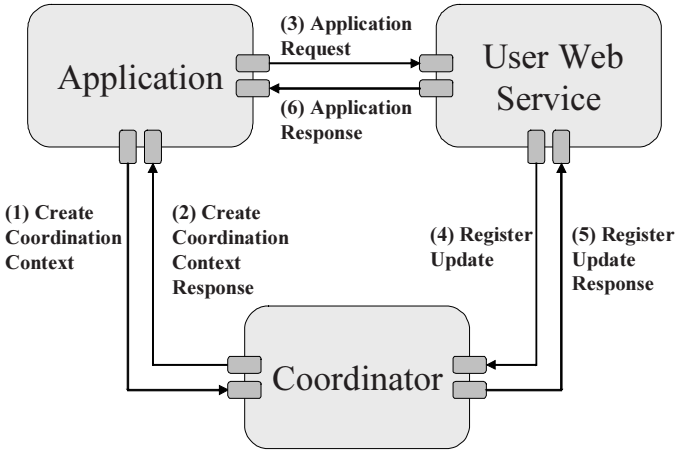

Fig. 3. Making an update to a Workspace through a Web Service

points could be defined for more complex environments, e.g. allowing data to automatically be updated between workspaces, automatic version check out options, merging between sibling workspaces, more flexible schemes for backing out committed operations, or creation of subsetted or time based workspaces. Note that the 'Application' in these illustrations may not be a web server application but may be any process capable of initiating web service calls.

The names of these operations and messages are relative to the base URL, for which we tentatively propose:

```
http://schemas.cs.ucc.ie/ws/2003/06/wsws
```

The coordination protocol provides for the creation and return of a CoordinationContext type, the propagation of that type on calls made to other services, and the registration of services by clients to receive notifications when certain user level operations are executed.

The following operations are operations of the Workspace Coordinator's Activation service. This port type is for use by applications to manipulate the workspace. The communication pattern is for making an update is illustrated in Fig. 3.

- CreateCoordinationContext Operation: When called to create a context with the wsws coordination type, it either creates a new workspace or looks up an old workspace depending on the arguments. The CreateCoordinationContext message includes the following elements:
  - WorkspaceName: The name of the workspace to be created or modified
  - ParentWorkspace: The optional name of the parent for the new workspace. If the WorkspaceName is new and this is not given, the new Workspace will be a child of the application default workspace.
  - User: The optional user name of the user performing the operation. This user is to be associated with any update made using this Context for auditing or undo.

In addition, if there are any access rules for this workspace, the user will have to meet those requirements. If the user is missing, there is no user associated with the context. The user being set here is a representation of the end user that is using the application either directly or indirectly, not the data store user that is used to make the connection to the data store. A typical web application runs using a single data store user per data store even while providing service to thousands of authenticated end users. The recipient of this operation may trust that its caller has already authenticated this user. Alternately the user name may be replaced with a signed delegation so that the coordinator knows that the end user has authorized the caller to act on its behalf.

The response to this operation is a CreateCoordinationContextResponse message that contains a CoordinationContext that should be sent passed on to all operations that are to act in the context of this workspace.

- Refresh Operation: Called with a WorkspaceCoordinationContext. Merges updates from the parent workspace into this workspace. Responds with a Refresh Response message which contains a ConflictList if there were any conflicts, that is, data items that were changed both in the parent and this workspace since the last successful Refresh. Data item versions that could not be merged are left unchanged in this workspace. While conflicts remain unresolved, the workspace may be in an invalid state and must not be published.
- GetConflicts Operation: Called with a WorkspaceCoordinationContext. Responds with the list of unresolved conflicts in this workspace, similar to that returned by Refresh Operation. Conflicts are created by the Refresh operation, above, and are resolved using the resolve operation below.
- Publish Operation: Called with a WorkspaceCoordinationContext. Merges this workspace into the parent workspace and returns a Publish Response message. If there are any conflicts, the operation fails making no updates to the parent workspace. If this happens, the client should perform the Refresh operation, resolve the conflicts, and then invoke Publish again.
- Undo Operation: Undoes the last update that was performed by this user in this workspace. If the user was not set as part of this WorkspaceCoordinationContext, it will undo the last update committed by any user in this workspace. If there were any subsequent incompatible changes to those objects, a ConflictList is returned as part of the response and the objects are left unchanged [24]. The undo operation never undoes an undo; it instead undoes the previous committed update. Use the Redo operation below, to undo an undo. Note that the changes instigated by the Undo and Redo must be committed before any other client can see them. This operation may also fail due to the data for the user's last operation being unavailable.
- Redo Operation: Undoes the effects of the last Undo operation done by this user in this workspace. Returns any conflicts caused by changes made to this workspace by other users that conflict with changes that the redo wants to perform. If there is any such conflicts, the Redo operation fails having made no changes. Note that undo and redo are most effective if they are done in a workspace that is private to a single user, as in that case no conflicts will ever arise.

A ConflictList object contains a list of data item Conflict elements. Each Conflict element contains a set of WorkspaceCoordinationContexts that when included with a web service request to read the given data item, allow all three versions of the conflicted object to be read, the parent version, the child version, and the ancestor version.

In addition the Conflict element provides a port reference to a service end point implementing the ConflictResolution port type. This port type provides an operation for marking the conflict as resolved.

- Resolve Operation: Marks this conflict as being resolved. This should normally be called after making any needed updates to the conflicted data item version in this Workspace.

5 Workspace Coordination Protocols

In addition to the application level protocols, additional port types are needed to allow the web services performing updates to communicate with the coordinator, to register their updates with the coordinator and to be called back in response to the high level application operations. A publish operation is illustrated in Fig. 4.

For this purpose we define the following port types. The Workspace Register port type supports the following operations:

- Register Update Operation: A specialization of the Coordinator Register Operation to allow an update made by a web service to be registered. The message includes a port reference to a port implementing the Update Manipulation port type. This information is used to implement the application level Undo and Redo operations. The Register Update message can contain arbitrary service specific elements that are also recorded, to identify the update and to make undo and redo of the update more efficient. The Register Update operation should either be made as part of an atomic transaction along with the actual update operation, or it should be done before the update and the implementation of the registered Undo Operation should be able to deal with the fact that the update may not actually have been performed.
- Register Data Store: A specialization of the Coordinator Register Operation to allow a data store used by a web service to be registered. The message includes a port reference to a port implementing the Data Store port type. The coordinator maintains stably the set of data stores used by the workspace so that they can be invoked to implement workspace level operations.

The Data Store protocol contains an operation for each of the Workspace wide application operations. In each case, these operations do the operation, but only on the subset of the data items residing on this data store.

- Refresh Operation: Merges updates from the parent workspace in this data store into this workspace. Returns a list of conflicts which the coordinator unions with the result of the Update operations on the other data sources.
- Get Conflicts Operation: Returns all conflicts in this workspace in this data store. The coordinator unions the result of this operation over all data stores registerd for the workspace.
- Publish Operation: Merges the changes made in this workspace onto its parent's workspace.

The Update Manipulation protocol has operations corresponding to the Undo operations and is used by the coordinator to perform user level undo operations. Note that the Register Undo operation

- Undo Operation: Undoes the indicated operation. Takes as an argument the Register Update operation that was used to log this event. If the Register Update and the actual update were not done as atomic transactions, then the implementation of this operation has to be ready
- Redo Operation: Redoes the indicated operation. Takes as an argument the Register Update operation that is to be redone. This operation can only be executed after an Undo Operation with the same argument has been executed in this workspace, or a workspace branched from it.

6 Relationships with the WS-Transaction Protocols

The WS-Transaction protocol [7] provides two sets of protocols, a tightly coupled protocol for atomic transactions and a less tightly coupled protocol for coordinating business activities. The workspace protocol proposed here lies halfway between these two protocols, matching the tightly structured format of the atomic transaction

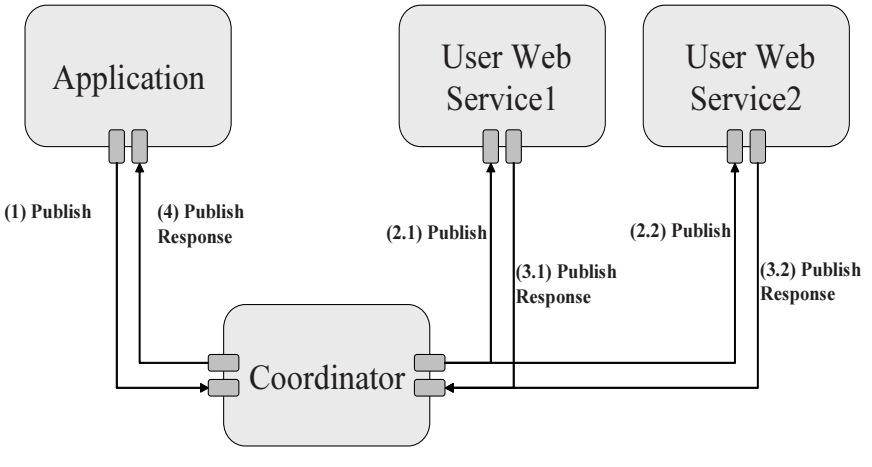

Fig. 4. Performing a publish operation

protocols, but supporting the long running, lock-free activities of the business activity protocol.

The three protocols solve different problems and are complementary. For example, one could layer the workspace protocols on top of the atomic transaction protocol. Each atomic transaction could then be used to bracket a set of changes, perhaps from different data stores, that make up a workspace update. This allows the scope of an update to be defined by the application rather than by each web service. This allows the notion of an update and thus of undo to be defined in an application specific, rather that service specific, way.

Similarly the business activity approach can be used to specify the required workflow to be used in publishing a set of changes made in a workspace, or to specify a compensation to be performed when undoing an update in a particular workspace that refers to data that is stored in an unversioned data store.

For certain data, architects will have to decide whether to model a particular user action as a compensateable update in the primary workspace, or as a regular change in a secondary workspace. The decision will typically hinge on visibility. The update in the secondary workspace will be effectively hidden, while the update in the primary workspace will be seen by all, even though it may later be undone by a business-process compensation.

Consider putting together an academic department's schedule of classes for next year. For some time the users might consider themselves to be in authoring mode and would be happy for there to be no actual rooms or instructors assigned to schedule. However at some point, the users want to start collecting real rooms and get approval for hiring additional instructors. By that point the itinerary would need to be published to a primary workspace shared by all of the departments and by the facilities department, which is responsible for assigning classrooms, and the finance department, which is responsible for approving budgets. Of course the academic department may continue to make changes to the schedule even after it has had classrooms assigned and job requisitions approved. In this case, the changes may invalidate the work done by the other department and may need to trigger business process actions, automated or otherwise, to reassign the resources.

7 Management Protocols

In addition to the protocols needed to implement the application level functionality, versioning systems have need of management protocols to support:

- Version browsing: Viewing the version tree of a particular data item.
- Auditing: Finding all the updates performed by a given user or in a specific time period.
- Workspace Access Control: In addition to having access control on individual data items, there can be access control on workspaces. Sometimes the access control on a workspace may override the access control on an individual data item. For example, a marketing researcher may not have write permission into the discount rules in the production workspace, but the researcher may have write permission

into these rules in a development workspace. Often a workspace may be locked so that no one can make any changes so that a particular snapshot of the environment may be saved
- Workspace browsing: Browsing and pruning workspaces. Unused workspaces can tie up lots of storage unnecessarily.
- Cache rule manipulation: Infrequently accessed data item versions may be represented by deltas from some base versions, so fast access means that these versions need to be cached.

8 Future Work

The author is evaluating this protocol and interfacing it with data stores that already implement workspace versioning. The implementation of the coordination protocol is similar to that of a transaction manager in its integration with the data stores. Versioned data stores that support the workspace model include Oracle's relational Workspace Manager [23], Microsoft's Repository object store [2, 3], and IBM's Clearcase file store [25], as well as various research efforts [19, 27].

While undoubtedly there will be many issues that will arise during the implementation of the system, the most interesting question to be answered is the usefulness of the distributed workspace model to application programmers. Just as the proof of a pudding is in the eating, the proof of a new data model is in the using. For this reason we will also be building several significant applications where the authoring, collaboration and data manipulation component are important. Applications and scenarios we are considering include:
- Catalogue and business rules maintenance in an e-commerce system. This example is examined in more detail in [28]. It involves a team building the autumn catalogue for an e-commerce system: inserting new content in the content management system, new products in the product database, new business rules in the rule system, and new accounts in the ERP system. The catalogue is constructed in a set of private workspaces and then merged into the main workspace on the 'go live' date.
- Work-flow maintenance. In this example, we devise a simple data driven system for routing and approving insurance claims. We then postulate that a revised claim processing workflow is to be implemented, tested and deployed. This change in policy is implemented by a member of the IT staff who tests it on a set of fake claims. The claims processing manager, after seeing who the new system in a mock up environment, agrees to try use it for all new claims initiated in the Seattle office, while existing claims and claims initiated in other offices continue to use the old policy. Eventually, after some tweaking and an ill advised change made to the Seattle test environment and then undone, the new workflow is released to the whole company.

Since this paper was written, the WS-CAF [4, 5] protocols have been introduced to complete with the WS-Coordination protocol discussed here. It appears that this work is largely independent of the underlying coordination framework and that WS-Workspace can be built on top of either framework.

9 Conclusion

As web services evolve to handle more complex problems, they will also need to support the authoring of more complex data. This will drive user requests for web services that can support full featured authoring environments for their data including features like

- Undo
- Batch Publishing
- Collaboration
- Auditing and Change Tracking

While it is possible for each web service to implement these notions in its own separate way, a coordination protocol allows the same approach to be used for all versioned web services and allows for workspaces and updates to span many different web services and their associated data stores.

We have introduced a coordination protocol that allows a simple way of manipulating workspaces and an application protocol that allows web services to allow their callers to control their workspace environment. The WS-Workspace protocol takes the lessons learned from configuration management in CASE and CAD products and makes it available to business applications by allowing the integration of disparate web services into a uniform workspace model.

References

1. BEA Tuxedo ATMI, http://e-docs.bea.com/tuxedo/tux80/interm/atmi.htm
2. Bergstraesser, T., P.A. Bernstein, S. Pal, D. Shutt, "Versions and Workspaces in Microsoft Repository," Proc. SIGMOD 99, pp. 532–533.
3. Bernstein, P.A., T. Bergstraesser, J. Carlson, S. Pal, P. Sanders, D. Shutt, "Microsoft Repository Version 2 and the Open Information Model," Information Systems 24(2), 1999, pp. 71–98.
4. Bunting, Doug, et al., Web Services Composite Application Framework (WS-CAF). July 2003. http://developers.sun.com/techtopics/webservices/wscaf/primer.pdf
5. Bunting, Doug, et al., Web Services Transaction Management (WS-TXM). July 2003. http://developers.sun.com/techtopics/webservices/wscaf/wstxm.pdf
6. Cabrera, Felipe, et al., Web Services Coordination (WS-Coordination). August 2002. http://www.ibm.com/developerworks/library/ws-coor
7. Cabrera, Felipe, et al., Web Services Transaction (WS-Transaction). Aug 2002. http://www.ibm.com/developerworks/library/ws-transpec/
8. Cellary, W., and J. Rykowski, "Multiversion Databases - Support for Software Engineering." Proc. of the 2nd World Conference on Integrated Design and Process Technology, pp. 415–420, Austin, Texas, 1996
9. Cheung, Susan, and Vlada Matena, Java Transaction API (JTA), Version 1.0.1, Sun Microsystems Inc., April 1999
10. Chou, Hong-Tai, Won Kim. "A Unifying Framework for Version Control in a CAD Environment." VLDB 1986: 336–344

11. Clamm, G., J. Amsden, T. Ellison, C. Kaler, J. Whitehad, Versioning Extensions to WebDAV (Web Distributed Authoring and Versioning). March 2002. RFC 3253. http://www.ietf.org/rfc/rfc3253.txt
12. Dart, Susan, Spectrum of Functionality in Configuration Management Systems, CMU/SEI-90-TR11.
13. Diaz, Angel Luis, Peter Fischer, Carsten Leue, Thomas Schaeck, Web Services for Remote Portals (WSRP). http://www.ibm.com/developerworks/library/ws-wsrp/
14. Goland, Y., J. Whitehead, A. Faizi, S. Carter, D. Jenson, HTTP Extensions for Distributed Authoring – WEBDAV, RFC 2518. February 1999. http://www.ietf.org/rfc/rfc2518.txt
15. Interwoven Inc., TeamSite 5.5, http://www.interwoven.com
16. Jomier, G., W. Cellary, The Database Version Approach, Networking and Information Systems Journal, Hermes Science Publishing 2000, Vol. 3, pp. 177–214, January 2000
17. Katz, R.H. "Toward a Unified Framework for Version Modeling in Engineering Databases," ACM Computing Surveys 22, 4 (Dec. '90).
18. Klahold, Peter, Gunter Schlageter and Wolfgang Wilkes, A General Model for Version Management in Databases, VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan
19. Marquez, A., Orthogonal Object Versioning in an ODMG compliant Persistent Java, Department of Computer Science, Australian National University, http://www.cs.adelaide.edu.au/~idea/idea7/PDFs/marquez.pdf
20. Microsoft Corporation, Microsoft Mappoint WebService. http://www.microsoft.com/mappoint/net/
21. Microsoft Corporation, Microsoft Transaction Server. http://www.microsoft.com/com/tech/MTS.asp
22. The Open Group, Distributed TP: The XA Specification, C193 UK ISBN 1-872630-24-3, February 1992
23. Oracle Corp. Oracle Workspace Manager, http://technet.oracle.com/products/workspace_mgr/content.html
24. Prakash, A. and Knister, M.J., "A Framework for Undoing Actions in Collaborative Systems," ACM Trans. on Computer-Human Interaction, Vol. 1, No. 4, pp. 295–330, Dec. 1994.
25. Rational Software. Rational Clearcase. http://www.rational.com/products/clearcase
26. Sciore, E., "Versioning and Configuration Management in an Object-Oriented Data Model," VLDB Journal 3, 1994, pp. 77–106
27. Soules, Craig A.N.. Garth R. Goodson, John D. Strunk, Gregory R. Ganger. Metadata Efficiency in a Comprehensive Versioning File System, May 2002 CMU-CS-02-145 School of Computer Science Carnegie Mellon University
28. Swart, Garret, Collaboration and Undo: The Web Workspace Paradigm, Fourth International Conference on Web Information Systems Engineering, 2003.
29. Vignette Inc., Vignette V7, http://www.vignette.com
30. WikiWeb, Web Based Collaboration Tools. http://www.wikiweb.com/