13

USING MULTIVERSION WEB SERVERS FOR DATA-BASED SYNCHRONIZATION OF COOPERATIVE WORK

Jarogniew Rykowski
Department of Information Technology
The Poznań University of Economics
Mansfelda 4
60-854 Poznań, Poland
e-mail: rykowski@kti.ae.poznan.pl

In this paper a new architecture for a multiversion Web server is proposed. This architecture is of three-tier type and consists in using an interpreter of a query language as a gateway for the server, specialized object-oriented multiversion database of resources as an engine, and an XML wrapper as a gateway for data repositories. The database-based server engine is capable of maintaining resources, their versions, and their meta-data. The engine performs also configuration management and consistency checking. The XML wrapper allows storing both data and meta-data across many repositories. Moreover, it performs caching and request redirecting for resources and their meta-data from mono- and multiversion servers. The proposed multiversion Web server may be used as generic information exchange tool for cooperative work in the integrated processes of development, production, and maintenance.

1. INTRODUCTION

In its early stage, WWW system – by now the most popular service across the Internet – was treated as a simple method for accessing remote, however static and formatted, mainly text-based documents. However, WWW system is growing rapidly, not only taking into account number of computers connected, but also by increasing a quality of its services. WWW system evolves – from simple, file-system-based, document management system to a universal application platform. It is done on one hand by growing possibilities of browsers (virtual reality, multimedia, new programming languages – Java and JavaScript), and on the other hand - by growing possibilities of servers: CGI scripts (CGI, 1999), servlets (Servlet), new protocols for exchanging WWW documents, e.g. – (XML, 2000).

At the very first look, due to great number of available services and popularity, the Internet is an attractive base for a cooperative system based on information exchange. First of all, the Internet is distributed. Its services are well scalable and can be fit to any application, from a local intranet server for a few people, to global systems like search services and public network portals for millions of users every day. There are several well-known and widely used communication services: e-mail,

news, file & data transfer, remote computing, etc. Web servers can store and give access to data of any type and size, offering in addition some utilities, like client- and server-side computing and data formatting.

However, while trying to build an integrated environment for a cooperating group of people based on currently available Internet services, several restrictions apply. The most important is no version support. A need for versions appears almost everywhere in the area of integrated processes of development, production and maintenance, to model: product customization to meet needs of local markets, different hardware and software used during development, production and maintenance, continuous changes in working team, different cultures, customs, and languages of people coming from and working in different countries, different time zones, etc. Moreover, during the integrated processes of design, implementation, and maintenance, information has to be represented in several forms specific for each stage of these processes. These forms may be generated manually (e.g., requirements, end-user manuals), semi-automatically on the basis of other forms (e.g., program skeletons generated from diagrams), or fully automatically (e.g., software binaries from sources by the use of compilers).

Taking into account the above characteristics of the integrated processes of development, production, and maintenance, one may say that a modern system for modeling these processes must at least: (1) represent the history of changes, (2) represent alternative ways of product development, (3) represent different forms of product components and ways of their transformation from one form to another, and (4) synchronize work of individual designers and the whole design team. To meet these requirements, different version types are required: revisions to model history of changes, and variants to model alternative ways of development as well as different forms of project components. Moreover, some mechanisms should be provided to separate and synchronize changes introduced asynchronously by designers. To this goal, using configurations as basic unit of consistency of the whole project would be good choice. A configuration is usually defined as a consistent set of versions of designed elements, one version per element. Configuration is a unit of consistency – one must always work in a scope of a given configuration. To synchronize work done by different designers in different configurations, such configurations could be merged, i.e., their versions could be inter-mixed to create new, better configuration being a result of cooperative work.

Up to now, no generic Web-based version support is available. Usually, only current moment of time is modeled across the network, and there is no simple and universal way of accessing different, for example past versions of Web resources. Some research activities towards universal version support for Web services are performed by WebDAV and Delta-V working groups from W3C consortium. These activities, however, do not address different version types: revisions to model changes in time and variants to model alternatives of some resources, as well as they do not propose strict configuration management for the whole multiversion system. More promising solutions can be found in the area of multiversion, object-oriented databases, but they are not well-suited to semi-structured, Web-based data. Smart merging of the above research activities looks to be a good choice.

In this paper a new architecture for a multiversion Web server is proposed. This architecture is of three-tier type and consists in using an interpreter of a query language as a gateway for the server, specialized object-oriented multiversion

database of resources as an engine, and an XML wrapper as a gateway for data repositories. The query language is used to define, manipulate, and control both data and their corresponding meta-data (data descriptions). The database-based server engine is capable of maintaining resources, their versions, and their meta-data. The engine performs also configuration management and consistency checking. The XML wrapper allows storing both data and meta-data across many repositories. Moreover, it performs caching and request redirecting for resources and meta-data located in other mono- and multiversion servers. The proposed multiversion Web server may be used as generic information exchange tool for cooperative development, production, and maintenance.

The rest of the paper is organized as follows. In Section 2 current approaches to version content of Web servers are described and discussed. In Section 3, the proposed architecture of multiversion Web server is presented, and basic functionality of the lower (physical) tier and the middle (logical) tier is pointed out, including query language and multiversion database. Section 4 concludes the paper and points main topics of future work.

2. VERSIONING CONTENTS OF WEB SERVERS

There are many investigations in the area of versioning in the Web, as well as in the area of database systems, the latter devoted mainly to CAD/CAM/CASE tools. In the framework of the W3C consortium there is a working group Delta-V (Delta-V, 1999), created as a spin-off from the WebDAV working group (WebDAV, 1999), devoted to authoring and versioning in the Web. This group clearly defined main goals and rationales for versioning in the Web (WebDAV-goals, 1999), and developed a prototype multiversion server. Main features of Delta-V approach are discussed below, with our proposals going beyond Delta-V work printed in italics.

There must be both transparent and apparent versioning provided. In addition to this requirement *we should add that non-versioning aware clients do not have to be informed they work in the versioned environment.*

It must be possible to version resources of any media or content type. In addition to this requirement, *way of storing, maintaining and accessing media of different types should be identical (e.g., uniform query language, access protocol, etc.).*

Versions, their configurations, etc. (*all the versioned and versioning resources maintained by the server*) should be named. Moreover, there must be a mechanism *for defining meta-data (descriptions) of resources. The meta-data should be user-defined.* It must be possible to use the same names and meta-data for different versioned resources. Configurations (workspaces in WebDAV terminology) formalize this relationship. Inside a given configuration, resource names should be unique. Given multiversion resource name and configuration name it must be possible to identify uniquely a given version (i.e., value) of this resource. Names *as well as meta-data* should be searchable.

Version history should be kept and be fully queryable.

The server must support the following operations:

- creating, *storing, maintaining*, and accessing versions of resources as well as their meta-data (name, label, author, creation date, comment, etc.),
- creating, *storing, maintaining*, comparing, and accessing configurations, *defined as consistent sets of resource versions, one version per resource*:

- the server should allow *user-defined* versioning policies;
- it must be possible, given a reference to a version of a versioned resource, to find out which versioned resource that version belongs to;
- the server should work in such a way as to minimize the adoption barriers for clients and existing repository managers. This includes integration with legacy data *in any repository, both mono- and multi-version*;
- it must be possible *to freeze a configuration and/or a version* (in WebDAV: to lock an activity) so that it is no more possible to make further changes;
- it must be possible to discover what resources have been changed from a given point of development (point of time), *by whom and why. The latter may be obtained by using meta-data.*

The Delta-V proposal has several limitations:

- multiversioning requires significant changes in existing protocols rather than building something above them. For example, for WebDAV servers, significant extensions to HTTP protocol are used to make this protocol multiversion. Changing protocols requires introducing changes both at server and client side;
- no distribution of multiversion objects is considered. Due to used centralized model of multiversion objects, information about versioning must be centralized or at least centrally managed. As a consequence, the proposed system is not scalable;
- no configuration management for all the objects and mutual interactions (links) among them is provided. Such management is restricted to a group of objects, while the rest is either not versioned or their versions are not fully synchronized;
- there is no distinction between modeling changes in time and alternate ways of development. To this goal, two types of versions should be provided. To model such changes in time, revisions are used. A revision of a multiversion resource is usually created on the base of another revision of the same resource as a result of a change performed to the latter one - revisions model resource history. Second type of versions, variants, are used to model parallel, alternative ways of development of a given multiversion resource. Variants allow solving some specific problems, concerning for example: different cultures and customs, different national languages, different time zones, different hardware and software, etc. For variants of a given group some operations can be performed, as it would be a single resource, for example "update all the versions of A written in German", etc.;
- change propagation problem is not avoided due to early-binding versioning (i.e., allowing pointing to versions of resources directly. Changing one element forces changes for all the elements somehow linked to this one);
- there is no uniform way of creating, storing, maintaining and accessing versioned resources and their meta-data.

Some of the above disadvantages have been resolved in the area of object-oriented databases. To deal with revision, variants, and strong configuration management, the MHD multiversion data model has been proposed (Rykowski, 1996). In this approach, several real world states are modeled rather than a single (the last) one. A notion of a multiversion consistency is introduced, as a consistency of a set of logically independent database versions called revisions. One *revision* represents one real world state. *Multiversion objects* are stored in the database in such a way that in every revision exactly one version of an object is stored. From the point of view of a user versions of objects are independent. From the point of view

of a system, however, there is a possibility of storing many equal versions as one *physical copy* - thus there is no redundancy in storing multiversion information. It is also possible to easily find all the revisions a given version of an object is used in. Inter-connections (links) between objects may be established. A link always points from a version of a multiversion object to other multiversion object. In this way a consistency of a database is independent of a structure of links among objects.

In the MHD approach, the version semantics is related to the database versions rather than to the single objects. The main advantage of this is the possibility of automatically maintaining consistency of object versions of the same and different types. In this approach two version types are distinguished: revisions and variants. First type of versions, the above mentioned revision, is used to model changes in time of an object. Revisions are represented in a single revision derivation tree. Second type of versions, *variant*, is used to model parallel ways of development of an object. A variant is any arbitrary chosen set of revisions. Version of an object is identified by its identifier and identifier of revision. It is worth to emphasize that only late binding is used to identify object versions. To identify right physical object version of a given multiversion object in a given revision, an association table is used. This table is composed of rows, each row pointing to a given physical version and a set of revisions this physical version is explicitly linked to. For revisions that are not pointed explicitly by an association table, the following *derivation rule* is used: "if a given revision is not explicitly mentioned in the association table, it shares physical object version with its parent from revision derivation tree". This rule is recursive. To express non-existence of an object in a given revision, special *nil* value is defined explicitly linked to the root revision.

Any operation on a given revision is not propagated to other revisions. Thus, changing any multiversion object in a given revision does not affect other revisions, even if they physically share the same physical object version (Cellary, 1990). Moreover, group operations are introduced to operate on sets of revisions (variants). In a group operation, a result is computed only once for a single revision and logically propagated to all other revisions from the set/variant.

The MHD model, however, do not address other important features, which should be provided in the cooperative Web environment:

- physical data storage, and data distribution; there should be a possibility to distribute multiversion resources, their versions, and their meta-data among many repositories and servers with different organizations, levels of accessibility, authorization, privacy, etc. Moreover, there should be a possibility of access read-only, maybe monoversion elements and treated them as a base for preparing multiversion, writeable, "private" elements;
- remote access; to facilitate access to multiversion resources and their particular versions, a query language should be provided. This query language should be based on standards used on one side in the Web environment (XML, HTML, HTTP), on the other side – on SQL (Fortier, 1998) and OQL (Catell, 1994) query languages widely used in the area of relational and object-oriented databases;
- uniform management of both resources and their meta-data;
- configuration types; to allow changing, there should be two types of configurations: working, in which changes are allowed, and archived (stable, read-only), in which changes are not allowed. Once derived, configuration (regardless

of its type) should be autonomous, i.e., changes made in its "parent" should not reflect any of its elements (and vice-versa).

3. PROPOSED WEB SERVER ARCHITECTURE

A new architecture for a Web server is proposed, compliant with recently proposed three-tier architecture for Web services (Carey, 1999). This architecture consists in using an interpreter of a query language as main gateway for the Resource Server, Resource Server Engine for managing versions of resources as well as meta-data, and XML Wrapper as a gateway for data repositories (Fig. 1). The query language is used not only to read server's data, but also to define, manipulate, and control both data and their corresponding meta-data (data descriptions). The data model for the Resource Server Engine is based on the MHD versioning model (cf. Section 2). The Repository Manager with the XML Wrapper allows storing both data and meta-data in many different repositories, including flat file system and a back-end database. Moreover, the Repository Manager performs caching and request redirecting for remote resources.

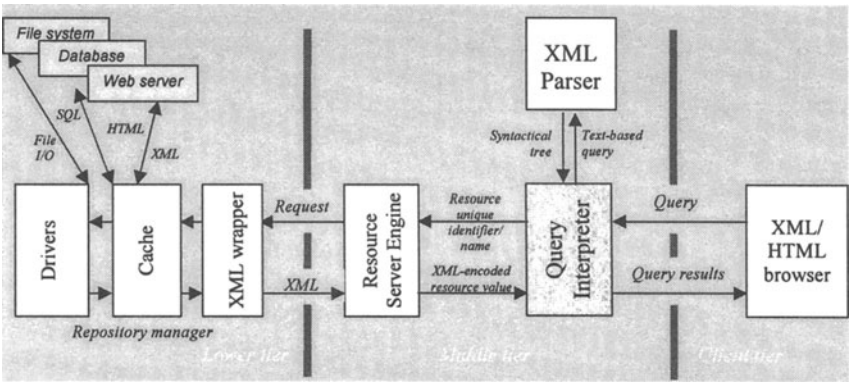

Figure 1 - The proposed architecture of queryable Web server

3.1. Lower tier – XML Wrapper as a base for repository manager

This tier is responsible for physical data access. The tier is composed of three layers: the XML wrapper, Cache layer, and Driver layer.

The way this tier functions is the following. A request from the middle tier (from the Resource Server Engine described below) is accepted by the XML wrapper. Based on the request parameters, mainly resource identifier (unique name), a resource is identified. For this resource, a check is made if it is already wrapped. If so, another check is performed if the wrapped resource value is still valid. The latter action stands for local cache of resources for the XML wrapper. If the resource is properly wrapped, its value is returned to the middle tier. If the resource is not wrapped, the Cache layer is contacted and a check is made if a raw resource value is already got from the external repository. If so, this value is sent back to the wrapper, where is properly encoded into XML, stored in local cache for the XML wrapper and finally sent back to the middle tier.

If the raw resource value is not cached, it should be get from a given repository. Looking into repository type, proper driver from the Driver layer is contacted to

fetch the value from given repository using given URL. Although usually the repository types and real resource URLs are system-generated, users may freely change them. Moreover, different resources may be physically stored in different repositories. There are several possible drivers and repository types, including all-in-memory driver, typical file-system driver, drivers for remote Web servers: WWW, LDAP, and Resource Server. The latter possibility is used for connecting Resource Servers among themselves, reflecting organization of a working team. Every member of the team uses its private Resource Server, with some resources accessed locally, some – exported to other members of the team, and some – imported from other Resource Servers (or other Web servers). This organization does not force the information to be localized and managed centrally, however, it makes it possible to keep all information always consistent, not only from a point of view of a single user, but also from the point of view of whole team. Note, however, that the consistency is restricted to single revisions rather than the whole distributed data (cf. description of the MHD versioning model in Section 2).

3.2. Middle tier - modeling revisions and variants in Resource Server Engine

Middle tier is responsible for data formatting. It is composed of three modules: Query Interpreter, XML Parser, and Resource Server Engine.

The middle tier functions in the following way. A text-based query sent from the client-tier is passed to the Query Interpreter. The Interpreter invokes standard XML parser (XML-IBM, 2000) (XML-MS, 2000) to build syntactical tree for the query. This tree is passed back and traversed by the Query Interpreter according to semantic rules of the query language. The interpretation process is recursive, i.e., if intermediate results are queries, they are processed in the same way. Thus, values of some resources or just computed results could be used as parts of other queries.

Note that, due to a fact that the Query Interpreter is the only input gate for the server, query language must be capable of creating, reading, manipulating, updating, and deleting resources. As presented in (Rykowski, 2000), current query languages are only capable of reading resources and, in a very limited way, creating new XML-based documents with search results. Thus, in comparison to the presented languages, the proposed query language is substantially extended by providing additional functionality for both searching and updating resources and their meta-data.

During query interpretation, values for resources are fetch from and stored into the Resource Server Engine. This module is responsible for management of resources, their versions and values, and their meta-data. To gain this, this module directly accesses the lower (physical) tier to read and write values of resources.

The Resource Server Engine is implemented as an object-oriented multiversion database, with well-defined schema, and instances of objects fetched from the lower tier – the XML Wrapper (cf. previous section). This module uses the MHD approach to versioning in object-oriented databases (cf. Section 2).

The database schema is presented in Fig. 2. Every object in a database is composed of a system-defined unique identifier, system-defined type, user-defined name and comment, and meta-data implemented as a set of text-based attributes (attribute list). Some objects are composed of sets of references to other objects. The meta-data are mainly user-defined, although some meta-data are predefined by the

system, for example: author's name, create and last access dates, size, etc. Note that although meta-data are kept outside resources, due to the MHD model and its strong configuration management they are always consistent with their corresponding resources. The inter-object references are based on system-generated unique identifiers of objects.

The main entry point is the multiversion database object. This object is composed of a set of revisions creating revision derivation tree, and a set of multiversion objects. Every revision is identified by unique version stamp. A revision is composed of a set of revisions being its direct children. Every multiversion object is composed of its association table. Every association table is composed of several rows, each association table row reflecting one physical object version and a set of version stamps of the revisions this physical version is directly linked to. For revisions that are not explicitly mentioned in the association table, the MHD derivation rule is used to identify right physical versions (cf. Section 2).

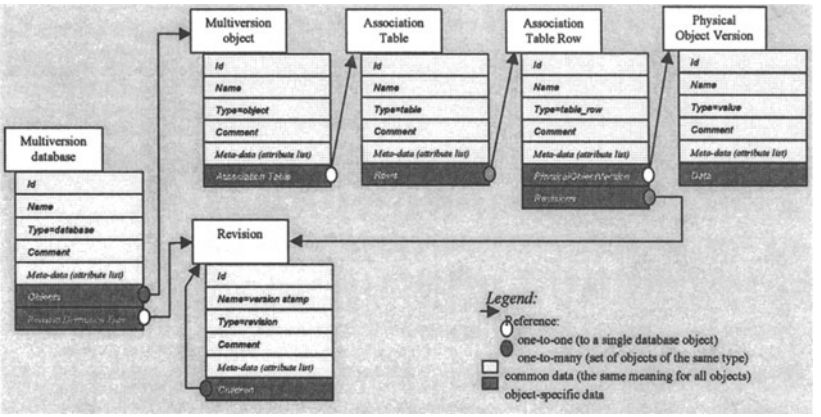

Figure 2 - Schema of the multiversion database of the Resource Server Engine

Above the schema, there are several operations for: (1) manipulating all objects regardless of the type, (2) manipulating revisions, and (3) manipulating multiversion objects and their physical values.

There are the following operations on multiversion objects:

- *reading* system-defined unique object identifier and type. Once set by the system during the creation of the object, these values cannot be changed. They are used by the database to identify the objects and references to them;
- *reading* and updating name of an object. For unique names, this operation is restricted to avoid ambiguity. For system-defined names (like version stamps for revisions), this operation is not permitted;
- *reading* and updating comment (modeled as a character string) on an object;
- *reading* and updating attributes from the attribute list of an object; for system-predefined attributes (author's name, dates of creation and last modification, value size - if any, etc.) updating is not possible, although reading is unrestricted.

There are the following operations on revisions:

- *creating* a new revision as a child of a given, existing revision. Newly created revision is automatically stamped by its parent version stamp and current child number. Revision version stamps are used as a unique names to identify revisions.

Note that creating a new revision does not force changing any data in the database, due to the derivation rule of the MHD approach. Newly creates revision is characterized by empty comment and a name expressing creation time;

- *deleting* a revision. The pointed revision is not physically deleted, instead it is only marked as deleted. Thus, complex operations on the revision derivation tree and the association tables of the multiversion objects are avoided. Moreover, at any time the revision may be undeleted without any modification of the database.

There are the following operations on objects and their physical versions:

- *creating* a new multiversion object. To create a multiversion object it is enough to register its unique name in the list of multiversion objects and create its empty association table. Empty association tables contain only one association table row pointing for all revisions to system-defined nil physical object version. Thus, newly created multiversion objects have empty values in all revisions. As for revisions, newly created multiversion object is characterized by empty comment.
- *deleting* a multiversion object. While deleting the multiversion object, its association table with all physical versions is also deleted. Once deleted, the multiversion object cannot be restored, although its unique system-generated identifier will never be used.
- *reading* a multiversion object value in a given revision. By consulting multiversion object name and version stamp of a given revision, a proper physical object value is fetched from the appropriate row of the association table of this object. Note that (1) the fetched value is always defined, but it could be nil "empty" value expressing non-existence of this object in this revision, and (2) there is one and only one physical value for a given multiversion object name and a given revision;
- *reading* multiversion object values in a given set of revisions. This operation is a group operation, i.e., it operates on a set of revisions and points out a set of physical values. The set of revisions could be identified either by a set of version stamps, or by an attribute list. In the latter case, only these revisions are taken into consideration for which attribute lists are subsets of a given attribute list. The latter is equivalent to varianted operation, as any set of revisions (identified either by common attribute list or given as a set of revisions) creates a variant;
- *updating* a multiversion object value in a given revision – this operation is similar to reading operation except physical values are updated rather than read;
- *updating* a multiversion object value in a set of revisions or in a variant – this operation and its parameters are similar to reading multiversion object values in a set of revisions or a variant, except the physical update operation is performed only once and the obtained result is logically propagated to all the pointed revisions or variant members (cf. group operations mentioned in Section 2);
- *deleting* a multiversion object from a given revision (set of revisions, variant) – this operation is equivalent to updating by the *nil* value (cf. Section 2).

3.3. Client tier – required browser capabilities

Due to a fact that queries and query results are XML-encoded texts, no special support is needed and any standard XML browser could be used. However, to facilitate operating on resources from the server, a dedicated Resource Editor is proposed working in visual mode and enabling creating, manipulating, reading, updating, and deleting resources, including both data and meta-data.

4. CONCLUSIONS AND FUTURE WORK

In this paper a new architecture have been proposed for a Web server compliant with three-tier model and capable of:

- creating, storing, maintaining, updating and deleting multiversion resources described by meta-data. The server engine is based on a specialized, multiversion, object-oriented database;
- using query language with exhaustive search possibilities as the main gateway to the server; the query language is based on well-known standards: SQL for querying database contents, and XML for exchanging data across the Web;
- searching for both data and meta-data;
- strong consistency checking for connections among resources and their meta-data. Even if meta-data are stored separately, they are always consistent with their corresponding resources. The same goes for different interconnections among resources;
- using different repositories, both local and distributed, e.g., flat file system, HTTP- and LDAP-based Web servers, relational database management systems;
- collecting data from different repositories and accessing them in a common, XML-compliant format by the use of the XML wrapper;
- using standard XML parser for interpreting the queries as well as for resource wrapping.

A prototype of a Web server based on the proposed architecture is now being implemented. The implementation is based on Java programming language. Current work is concentrated on the Query Interpreter and the Repository Manager implementation, to supply drivers for most of the Web services (HTTP server, LDAP server, FTP server), and local and remote relational database management systems like Oracle or any ODBC-compliant database.

5. REFERENCES

1. Carey M., eds. Post-Modern Database Systems: Databases Meet the Web, Berkeley University Course CS 294-7, http://db.cs.berkeley.edu/postmodern/
2. Cattell R.G., eds. The Object Database Standard: ODMG-93. Morgan Kaufmann, San Francisco, California, 1994
3. Cellary W., Jomier G.. Consistency of versions in object-oriented databases. Proceedings of the 16th Very Large Databases Conference, Brisbane, Australia, 1990
4. CGI scripts overview. http://hoohoo.ncsa.uiuc.edu/cgi/overview.html
5. Delta-V working group home page. http://www.webdav.org/deltav
6. Fortier P.J. SQL 3:Implementing the SQL Foundation Standard. at http://www.amazon.com
7. Rykowski J., Cellary W.. Multiversion Databases - Support for Software Engineering, proceedings of the 2nd World Conference on Integrated Design & Process Technology, Austin, USA, 1996
8. Rykowski J.. Architectures for Querying Contents of Web Servers, In proceedings of the 4th Int. Conference on Business Information Systems BIS'2000, Poznań, Poland: Springer-Verlag, 2000
9. Servlet documentation. Java Web Server, http://java.sun.com/products/webserver/features/
10. WebDAV working group home page, http://www.ics.uci.edu/pub/ietf/webdav
11. WebDAV working group goals, http://www.ics.uci.edu/pub/ietf/webdav/versioning/draft-ietf-webdav-version-goals-01.txt
12. XML documentation, http://www.w3.org/TR/1998/REC-xml-19980210
13. XML parser for Java, IBM, http://www.ibm.com/developer/xml
14. XML parser for Java, Microsoft, http://msdn.microsoft.com/downloads/tools/xmlparser/xmldl.asp