Design of a New World-Wide Information Storage System (Unfinished Second Draft 2a)

Robert L. Walton*

April 19, 2000

Contents

*This work was done with the aid of Dan Stefanescu, Aaron Bronfman, and Thomas Cheatham.

1 Introduction

The introduction of hypertext web pages in the 1990's created a world-wide information system called the WEB. The WEB is based on HTML documents that form an index to information stored in many forms.

Here we explore an alternative to HTML documents as a basis for a future world-wide information system. We will call our system WBW, which stands for 'world beyond the web', and suggests loosely that we are trying to make progress in storing world-wide information.

In addition to storing world-wide information, we also wish WBW to take over roles performed by operating system shell and code management tools, so that in the future these will be integrated with the world-wide information system.

The main goals of this paper are to lay out a design for the WBW storage system and to describe the critical choices made during the design and our rationale for making these choices. The WBW storage system is the bottom layer of WBW that provides basic ways to store and organize information.

The first new characteristic of the WBW storage system is that it adds security features normally associated with file systems, such as access control lists, to the world-wide information system. More specifically, the WBW storage system provides the security and reliability of local information access that one would expect from a modern file storage system, while also permitting much less secure and reliable global information access using the same logic and interface as for local access. *2a revised paragraph*

The second new characteristic of the WBW storage system is that it stores serialized JAVA objects, and not just passive information. JAVA objects provide the following capabilities: *2a new paragraph*

(1) New interfaces can be added to existing objects by writing and exporting new code.

(2) New object types can be made to support both new and old interfaces by writing appropriate interface code.

(3) Objects are not merly passive but can compute. For example, they can map inputs to outputs using both the contents of the object and a computational algorithm. As another example, they can map requests for information to remote procedure calls that invoke services anywhere on the internet.

(4) JAVA objects can map a single interface to many different implementations. For example, we can have a 'file' interface that provides ways of acquiring a file. Many different types of JAVA object may implement the file interface. These may or may not maintain caches of particular files, may copy files from one operating system to another or convert files from one format to another, and may even create files on the fly from rawer data.

The third new characteristic of the WBW storage system is that it permits all its users to create new types of active objects that process and save information. Thus anyone who can *2a new paragraph* post information on the internet can also create services that process information and store new information.

2 The Base Design

WBW stores information in a labeled directed graph. There are two kinds of nodes in the graph: object nodes and name server nodes. An object node can store a serialized JAVA object. Name server nodes cannot store anything.

Graph nodes are connected by directed arcs called links. The links are labeled with ASCII character string names. The nodes are not labeled.

Each object node is stored in a single WBW server, and in this sense has locality.

Many users are exposed to only one name server node: the 'universal node'. The links from the universal node are labeled with WBW server names, and target a server root node in each server. Some users may be able to access company-wide or project-specific name servers, and may not be able to access the universal node.

The graph can be cyclic. Links may connect object nodes in different servers. Several links to or from a node may have the same link label. Aside from the specialization of name server nodes and the fact that nodes do not have labels, the graph is a completely general labeled directed graph with no special restrictions.

Clients have open nodes. They can open a new node by giving a pathname relative to a node they already have open. A path can go both 'down' a link, in the direction of the link, or 'up' the link, in the direction opposite to the direction of the link. For example, the path `w/x/..y/z` starts at some source node, goes down the `w` link, then down the `x` link, then up the `y` link, and then down the `z` link, to some destination node. Here we used '`/`' to separate the arc names of a path and '`..`' to denote that a link is to be traversed in the 'up' direction, rather than the usual 'down' direction.

Clients must be initialized to have at least one open node, or else they cannot open any nodes, since a client gives an already open node and a pathname to open a node. Later we will see that clients can be trapped in subgraphs called sand boxes for security reasons. Most clients that are not in a sand box start with the universal node open, to play the role of the 'root directory', and a second open node playing the role of the 'current directory'.

Name server nodes have representatives that are nodes stored in WBW servers and are therefore local. A link to or from a name server node is implemented as a link within a WBW server to or from the WBW server's representative of the name server. The name server, which is user code as far as WBW is concerned, merely takes a pathname representing a path from the name server node and selects a WBW server to continue tracing the pathname. When a path being followed comes to a name server node representative, the name server designates a possibly different WBW server, the path hops to the name server node representative on the designated WBW server, and the path is continued on that server. In order to ensure security, WBW only permits name servers to hop between WBW servers. *2a new sentence*

Name server node representatives must store information on how to access the name server, including security information, such as public keys of the name server. Most WBW servers have a representative of the universal node.

When a client opens a node it gets certain permissions on the open node, as will be discussed later. Depending upon having suitable permissions, the client may read or write the serialized JAVA object at an object node, and read, make, or break links between nodes.

WBW clients load the serialized JAVA objects and run their methods. Each client uses JAVA class code signed by code sources trusted by the client. Except for some bootstrapping class code, all class code can be accessed just like any other information in WBW.

WBW servers do not run JAVA code and need not be written in JAVA.

The serialized JAVA objects are intended to store information that is most often read-only or accumulative. That is, it is not anticipated that the JAVA objects themselves will have changing state. The JAVA objects can serve as pointers and interfaces to non-WBW objects on the internet that do have state. For example, a JAVA object could point at and interface to a database server, so that operations apparently on the JAVA object were really executed remotely on the database server.

Accumulative information is information added to an object in such a way that if several users independently add information, all this information can be merged. Several users can independently read a JAVA object from an object node, each independently add information to the object, and then asynchronously write the object, with the expectation that the information they separately add will be merged. This is implemented by merging any updated object with the current version of the object. Merging is done by a type specific merge method run in a JAVA virtual machine sand box by the WBW server that stores the object.

The WBW storage system implements very limited searches directly. Such searches find a list of object nodes by checking link names and access control lists of links and nodes. *2a new paragraph*

The WBW storage system does not directly implement seaches based on object contents. Instead, the WBW storage system implements '*spigots*', which are output steams that begin with a copy of a subgraph of the WBW storage graph, and subsequently include updates to that subgraph. A search engine external to the WBW storage system can open a spigot, build *2a new paragraph* an index to the subgraph from the contents of objects in the initial copy of the subgraph, and maintain its index as it receives object updates from the spigot.

In order to permit subgraphs to be readily defined, the WBW storage system allows links to be marked with '*support flags*'. A subgraph can be defined as all nodes reachable from one or more given root nodes by following links marked with a particular support flag. *2a new paragraph*

3 Base Design Choices and Rationale

Below we will describe various extensions to the base design, but first we describe the critical choices made to derive the base design and the rationale we used to make these choices.

3.1 Using JAVA Objects

We have chosen to store serialized JAVA objects, rather than, for example, XML documents. XML documents provide a flexible way to store data objects with attribute labels and values, but unlike JAVA objects, XML documents have no associated code.

We chose to store JAVA serialized objects because of their capabilities which we listed near the end of section 1.

A major disadvantage of using JAVA objects is that we become entangled in code security concerns. Our current goal is to solve these as well as possible. Even a simple solution such as only loading code signed by trusted sources seems to provide more capability than data structures like XML that make no provision for code at all.

The current WBW storage system design does not generally depend upon the serialized JAVA objects stored in object nodes, and therefore the entire WBW storage system design could be easily changed to store XML documents instead, or to store either a serialized JAVA object or an XML document or both at each object node.

3.2 Using an Unrestricted Labeled Graph

Any kind of computerized information can be represented in a fairly convenient way as a labeled directed graph. We have chosen to structure WBW information as a labeled directed graph with almost no restrictions or additional structure. The only departure we have permitted from a pure labeled graph is that we have distinguished special name server nodes.

In particular, we permit nodes that store objects to be the sources of links, so no node in the labeled graph need be a leaf. We have no reason not to permit an object node to be a source of links, and it is likely to be convenient to link an object directly to objects to which it is related.

We also permit WBW storage graphs to be cyclic. In order to make it easy for the user to specify subgraphs, we permit links to be marked with support flags, so a subgraph can be defined as all nodes reachable from a given list of root nodes by following links with a given support flag. $2a_{new}$ *paragraph*

3.3 Labeling

Labels in a directed graph can be use both for naming nodes and for searching for nodes. Labels can be attached to either links or objects or both. Labels can be simple character strings, or can be attribute name/value pairs. A link or object may have at most one label, or may be permitted to have a whole set of labels. $2a_{new}$ *section*

For the WBW storage system we have chosen to use labels for relative naming of objects and not, for the most part, for searches. This leads to placing single character string labels on each link, and no labels on objects.

To support searches on the contents of objects, spigots are introduced to pipe the required information about a subgraph to a search engine, which can then organize this information in a manner appropriate to the searches it supports. This is much more flexible than trying to build a search scheme into the WBW storage system itself.

The WBW storage system does permit simple searches that use the labels it does have to find sets of similarly named objects.

3.4 Distinguishing Local and Global Servers

There are two kinds of servers, which we will call local and global. A local server is made of parts that are connected by fast reliable communications. Also, since an intruder who can physically access any computer on which part of a server runs can compromise that server, all parts of a local server must be on computers whose physical security are equal. However, a local server in our sense does not have to be geographically local: a bank, for example, could run a local server with a part in each of three main offices in different cities.

Global servers, by contrast, may not be coupled by fast reliable communications or may not consist of parts that are equally secure. Therefore the security of a global server may be uncertain.

Unlike other distributed systems that have been proposed[LEG, GLO], we insist on distinguishing between local and global servers. We do this to localize reliability and security issues. The WBW information graph is world-wide. Any client starts from one or more nodes in the graph, and travels by paths from these starting nodes to other nodes. We desire that if the starting nodes of a client have a certain level of security and reliability, and if the client uses only paths to nearby portions of the WBW graph with the same level of security and reliability, then the client will have this level of security and reliability.

We expect WBW servers to be local, so the reliability and security for clients that stay within one WBW server will be determined by that server, and will typically be high. A WBW server stores object nodes and links between these nodes, thus permitting these objects and links to be managed securely.

On the other hand, WBW name server nodes are intended for use with name servers that are usually global servers. Their use typically involves acceptance of less reliability or less security or both.

However, it is possible to implement a name server that is local, e.g. within a company, and is therefore more secure. Similarly it is possible to implement a WBW server that is global, and therefore less secure, though the result might not have the reliability users expect of a WBW server. Lastly, it is possible to link objects in different WBW servers, but programs that use such links must expect reliability and security to be that of the least reliable or secure WBW server or communications link used.

3.5 Object Links Between WBW Servers

We could forbid direct links between objects stored in different WBW servers, insisting that to get at objects in different WBW servers a client must use paths going through the universal node or perhaps some other name server. As a result the WBW graph would have a more tree-like structure.

We do not do this because historically early UNIX tried to get along with just a tree-like structure and UNIX user's found this too restrictive.

We could, like UNIX, introduce 'symbolic links' that just store pathnames. Unlike links between WBW objects, the destination of a symbolic link does not know the link exists. We do not use symbolic links because we want destinations to know of links for security reasons described in 4.2, 5.1, and 5.5.

A second reason we do not use symbolic links is because WBW paths can go both up and down WBW links, but paths can not go up symbolic links. If a complex information structure usually stored on a single WBW server made use of the ability to go up links, and then someone wanted to distribute the information structure across WBW servers, we do not want the apparent nature of the links to change, so we want to retain the ability to go up a link between two WBW servers.

3.6 Object Handles

Several distributed information systems[LEG, GLO] implement object handles that are sequences of numbers identifying an object, independent of any directory structure. A handle may be thought of as the name of a server followed by a number that identifies the object within the server. Directories that map ASCII character string names to object handles are then left to the user.

We have chosen to make directories part of the WBW server database so we can analyze directory information and use directory information to manage storage and security. So, for example, we can garbage collect all nodes not reachable from designated permanent nodes in a WBW server using support links within that server. And we can create subgraphs we call sand boxes (see 4.2 below) in which computations may be trapped.

Given that the directory structure is part of the WBW server database, there is no need to have user visible-handles. Handles do exist as part of the internal implementation of WBW servers, and can be read by users so that user's can analyzed the structure of the WBW information graph, but object handles cannot be used by users to to open or otherwise access objects.

Our decision to keep handles internal, except for information graph analysis purposes, is exactly the same as the decision made by UNIX, where file handles consist of a file system identifier and a within file system identifying number called an 'inode number'.

3.7 Keeping Code Out of WBW Servers

With the limited exceptions of name servers and merge methods, WBW servers run no code but their own server code. This is to facilitate security and administration of data. Thus the WBW storage system presents a pure database, without any customizing user code, that can be administratively analyzed and secured. If customizing user code were allowed, administrative functions and security would become impractical.

Interfaces that can be customized by user code can be built on top of the WBW storage system.

3.8 Moving Objects

Some systems permit objects to move from one server to another. Below we will give an extension to our basic design that will do this in a simple way with some disadvantages.

It is hard to make a general system for permitting objects to move on a worldwide scale. If you use global name servers to convert location independent object handles (3.6) into current object locations, you run into the reliability and security problems of global servers mentioned in 3.4. We wish to avoid this, and therefore give an extension to the basic design below that avoids using global servers, at the cost of hopefully infrequent inconvenience to the user.

In WBW a user can always create his own name server to locate objects. Such name servers can either be restricted to a few projects or to particular kinds of objects, and therefore have a much better chance of being reliable and secure than a global name server that maps all the objects in the world.

Also, WBW serialized JAVA objects do not have changing state, aside from accumulative state maintained by merging. Thus WBW JAVA objects are designed so that making multiple copies of the objects does not compromise object semantics.

3.9 Trusted Code

The history of computing is full of failed attempts to secure things like operating systems and JAVA execution engines[MF99]. The notion that the only secure way to run foreign code is in an operating system user process that is denied the right to make any operating system calls at all is very close to the truth. The sand box that a WBW server uses to run object merge methods will hopefully be a user process that cannot execute operating system calls.

Thus it is likely that many WBW users will only want to run trusted code. This is a bit sad, since it inhibits experimentation. If JAVA execution engines are found to be safe, a more lenient policy might be devised.

3.10 Information Mismatches

A WBW server and a client, or two WBW servers, may not assume the information they have matches, as due to crashes and restorations or due to one side being malicious the information may not match. Storage integrity and security must be maintained in the face of such information conflicts.

The design of the WBW storage system partitions information that must match into very small independent sets of information. Thus the information two WBW servers keep about a link between the WBW servers only affects that link. And the information a client and a WBW server keep about an open node only affects that one open node. On each case the WBW server or client must keep enough information to defend itself against errors by the other side of the link or other end of the open node.

4 Extensions of the Base Design

In the following subsections we describe various extensions to the base design. Then in the next section we describe some of the choices and rationale that lead to these extensions.

4.1 Access Control Lists on Objects

A WBW access control list, or ACL, is a mapping from public keys to yes/no permissions. Each permission may be granted by an ACL to everyone, or to no one, or to clients that possess private keys that correspond to public keys listed in the ACL for the permission.

For each permission an ACL implements a boolean function that grants the permission. This function is constructed from tests that a client has a private key, and from the logical 'and' and 'or' operators. Special cases of this function always map to true or always map to false. The 'not' operator is not permitted in the boolean function.

The 'not' operator is not permitted in a boolean function because we need the boolean function to be 'monotonic' in the set of private keys held by a client. In other words, if client X holds all the private keys of client Y and maybe more private keys besides, then the boolean function must be true for X if it is true for Y. If the boolean function were not monotonic, a client could get more permissions by pretending it did not possess a private key.

Each object node has an optional ACL.

When an object is opened by a client, the open object is given the permissions allowed the client by the object's ACL and the private keys possessed by the client. This rule may be modified by placing ACLs on links, as described in the next section.

In consulting the private keys of a client, a WBW server does not actually see the private keys. For example, the WBW server may choose a random number, encrypt it with the client's public key stored in an ACL, and send the encrypted random number to the client. The client decrypts the number using its private key, reencrypts it with the WBW server's public key, and sends it back to the WBW server. The WBW server then decrypts this message with its private key, and compares the result with the original random number: a match indicates the client possesses the private key that matches the public key in the ACL.

The following are standard permissions:

read	The serialized JAVA object at the node can be read.
write	The serialized JAVA object at the node can be written.
link-source-read	Links sourced at the node can be read.
link-source-write	Links sourced at the node can be created or deleted.
create	An new object node can be created on the same WBW server as this node.
link-target-read	Links targeted at the node can be read.
link-target-write	Links targeted at the node can be created or deleted.
ACL-read	The node's ACL can be read.
ACL-write	The node's ACL can be written.

If an object has no ACL, all permissions are granted to everyone.

In order to read or write an object, a client must first open the object with read or write permission. In order to read or write an object's ACL, a client must first open the object with ACL-read or ACL-write permission. In order to create or delete a link, a client must both open the source of the link with link-source-write permission and open the target of the link with link-target-write permission.

In order to create an object node, a client must first open another node on the same WBW server with create permission. Usually this other node also has link-source-write permission and the client makes a link from this other node to the newly created node.

4.2 Access Control Lists on Links

Two ACL's can optionally be placed on a link. The 'down-ACL' can be used to deny permissions for objects opened by using a path that takes the link in a downward direction. The 'up-ACL' can be used to deny permissions for objects opened by using a path that takes the link in the upward direction. The set of denied permissions in both cases is computed by using the ACL and the private keys held by the client to compute what permissions the ACL denies.

With the addition of ACLs to links, open objects are given two sets of permissions: link permissions and object permissions. This is in distinction to the system of the last section, where there were no link ACLs and open objects had no link permissions.

When an object Y is opened by a client, another open object X and a path from X to Y must be given. The open object Y is given the link permissions of X that are not denied by the path from X to Y. When a path traverses a link in a given direction, down or up, the link ACL for that direction and the private keys of the client are consulted, and may deny link permissions.

If the permission to read links in the direction next required for the path becomes denied, the path cannot continue, and fails for lack of permissions.

The object permissions of an open object are those allowed by the object's ACL and the private keys of the client, and not denied by the link permissions of the object.

Link permissions are used to differentiate between clients that are outside or inside a subgraph of the node/link graph. Link ACL's may be set to deny permissions to a client who starts outside or inside the subgraph and traverses a link between the subgraph and the outside world.

One application of link ACLs is to build a firewall between a set of computers inside a company and the outside world. One can arrange that clients inside the company can read the outside world but not write it, while clients outside the company cannot access WBW servers inside the company.

Another application of link ACLs is to run a client that might execute dangerous code inside a small subgraph so it cannot access, or merely cannot write, outside the subgraph. In this case the subgraph is called a sand box. It is usually desired to sand box a client in part of an existing computer, and because existing operating systems do not support sand boxing well, it is currently difficult to do effective sand boxing with a WBW client who can execute operating system calls. The WBW storage system does permit sand boxing for clients who only execute JAVA code in sand boxed WBW clients, and make no operating system calls.

The following additional permissions are defined:

link-source-ACL-read	The ACLs of links sourced at this node may be read.
link-source-ACL-write	The ACLs of links sourced at this node may be written.
link-target-ACL-read	The ACLs of links targeted at this node may be read.
link-target-ACL-write	The ACLs of links targeted at this node may be written.

If a link has no ACL in a given direction, the link denies no permissions when used in that direction. This would be the default situation on links that do not cross a firewall or a sand box boundary.

When a link is created, the client creating the link must open both the source and target nodes of the new link. The down-ACL on the new link must deny all permissions that the link permissions of the open target deny. The up-ACL on the new link must deny all permissions that the link permissions of the open source deny. This rule must be enforced by the WBW servers to ensure a client cannot create link permissions for the client and an object node.

To read or write a link down-ACL or up-ACL a client must open the link source with link-source-ACL-read or write permissions, and also open the link target of the link with link-target-ACL-read or write permissions. When down-ACLs or up-ACLs are written, the rules of the last paragraph are enforced relative to the link permissions of the open source and target.

When a name server causes a path being followed to hop from a name server node representative in one WBW server to a name server node representative in a possibly different WBW server, the name server may consult the client private keys and deny permissions.

4.3 Moving Object Nodes

Depending upon how the serialized JAVA objects supported by the WBW storage system are used, there may be little reason to move WBW object nodes. But it is possible to implement a service that does so.

A set of object nodes is best moved together. When this is done, all links to the nodes must be adjusted to the new location of the nodes. If there is a link from WBW server X to an object in WBW server Y, and that object is moved to WBW server Z, then if X and Y are not able to communicate when the object is moved from Y to Z, Y must remember that the object has been moved and must inform X of the move whenever X and Y are again able to communicate.

The link from X to the object that was in Y cannot be used without X and Y being able to communicate after the object has been moved from Y to Z. This is a potential inconvenience, as it may be desired to take WBW server Y down permanently after moving objects from Y to Z, yet if Y never communicates with X, the link from X will be broken.

Links between a name server representative in WBW server X and a node being moved from X to WBW server Y are usually switched to be to or from the representative of the name server node in Y.

4.4 Using Link IDs and Object IDs

When a WBW server first creates an object or a link, it assigns the object or link a unique numeric ID that is never again reused for any other object or link, even in the future. The object IDs cannot be used by the WBW user for any purpose other than tracking the history of the part of the graph maintained by a WBW server.

Link IDs can also be used in pathnames in place of link labels. They are most often used when the link labels are not unique. Thus a client can open an object, read every link sourced at that object, and then using link IDs open objects that each of these links target, even if these links do not have unique labels.

4.5 Audit Trails

A WBW server remembers links that previously existed but have been deleted, and link up and down ACLs that previously existed but have been updated. Using this historical memory, the state of the WBW server at a time past can be checked for administrative and security purposes.

Optionally, previously existing object ACLs, and previous values for the serialized JAVA objects stored in object nodes, may be remembered.

There may be a limit on how long historical information is remembered. Historical information is not generally available to clients: it is available to administrators.

4.6 Object Signatures

An object node can optionally store serialized JAVA object signatures, in addition to storing the serialized JAVA object. The signatures indicate agents who are responsible for the correctness of the current version of the serialized JAVA object.

Each signature has a protocol. For the simplest protocol, the signature is made by taking the MD5 hash function of the serialized JAVA object and encrypting it with the private key of the signing agent. That the object has been signed by a particular agent can be verified by decrypting the signature with the public key of the agent and comparing the result to the MD5 hash function of the serialized object.

4.7 Code Agent Identities

An object node can store the identities of agents who must sign class code for any serialized JAVA object stored in the object node. This insures, for example, that any object merged with the current serialized JAVA object will be merged by the proper class code.

4.8 Crash Recovery

WBW servers and their clients need to be able to recover from crashes.

One ideal for this would be for each side of a client/server communications connection to keep enough historical information so that if the other side crashed and was restored at a checkpoint in the past, the side that did not crash could undo previously done operations until it was at the same execution point as the restored side.

There are two problems with this approach. First, clients are unlikely to be able to back up, because keeping the required information for an arbitrary client program is too complex, since client programs can do arbitrary computations. Second, clients may interact with each other through the WBW storage system, so backing up the operations for one client may upset other clients.

Still, something can be done. If a client keeps a record of operations, and the server crashes and is restored as of a time in the near past, the client can replay any lost operations to the server, with a good probability of success.

Similarly if the server keeps a record of operations and responses, and a client crashes and is restored as of a time in the near past, the server can check whether the restored client is issuing the same operations as in the past, and replay responses, or if the client diverges from the past, the server can back up a few operations. This also has a fair chance of success.

WBW servers should keep dynamic checkpoints by writing their transactions to stable storage. In order to be efficient, a transaction client may be told the transaction is complete shortly before the transaction has been written to stable storage, resulting in the server's appearing to backup up in time if it crashes after notifying a client of the completion of a transaction but before writing the transaction to stable storage. However, the backup times for WBW servers should be very short unless something destroys the stable storage. WBW servers may also use shadow servers as stable storage: the shadow server mimics the operation of the WBW server and if the former fails, the shadow immediately takes over.

4.9 Garbage Collection

If some nodes stored by a WBW server are marked as 'permanent', the WBW server may adopt the policy of deleting all nodes not reachable from its permanent nodes by links that are inside the WBW server (between nodes on that server).

It is also possible for a small set of WBW servers to agree that only nodes not reachable from one of their permanent nodes by links within the set of WBW servers are to be deleted.

5 Design Extension Choices and Rationale

In this section we describe some of the critical choices used to derive the above extensions to the base design, and the rationale we used to make these choices.

5.1 Including of ACLs on Objects and Links

ACLs on objects are needed to control who can write an object, and in some cases, who can read the object.

However, the distinction between users inside a large group with a great many objects, like a company, and users outside the group, is difficult to make with object ACLs, because there are so many objects whose ACLs must be set correctly. It is easier to control the connections between the objects inside and the users outside. This can be done by placing up and down ACLs on links.

The same argument applies to constructing a sand box in which a complex computation involving many objects that may be created during the computation is to be trapped. It is easier to control the links between the sand box and the outside world than it is to control the objects inside the sand box.

Given that links have ACLs, it is possible for a user X with extensive permissions on a node O to grant a subset of these permissions to a user Y by making a link from a node accessible by Y to the node O that denies all permissions not in the subset. This is a powerful user tool for managing security, provided it is supported by other tools that examine the WBW graph and look for permission leaks.

5.2 Including Audit Trails

To be sure a firewall or sand box has worked properly, it is necessary to investigate not only current links and their ACLs, but also links and link ACLs that existed in the past. Thus we provide for keeping a history of links and link ACLs.

A history of object node values and ACLs may also be of security interest in some cases.

5.3 Security of Existing Operating Systems

One complaint about the WBW storage system design is that it does not interface with existing operating systems in a way that improves the security of these systems. But we have decided that it is unreasonable to try to improve the firewall or sand boxing capabilities of existing operating systems without redesigning these systems or replacing these systems.

One option is to add files to the object nodes of WBW and use WBW to replace the file systems of existing operating systems. Its not clear whether it is better to try to replace existing file systems in this way or to try to improve existing files systems.

5.4 Responsibility Tracking in Audit Trails

It would be very useful to record who is responsible for actions recorded in an audit trail: e.g., who is responsible for making or breaking links. The responsible person is not merely the client who executed the programs that made or broke the links, but also the authors of the programs that were executed. We have not yet determined how to add responsibility tracking to the WBW storage system.

5.5 Malicious WBW Servers

As the security of some WBW servers may be compromised, it is necessary for each WBW server to defend itself against every other WBW server.

For example, links between WBW servers must be defended. The WBW server on each side of the link must keep a copy of all information about the link, including the down-ACL and up-ACL. The link represents the granting of permission for a client of one server to open a node on the other server. If one server remains secure but the other server is compromised, the client who made the link is responsible if bad things happen at the secure server's node because of actions by the compromised server. But the secure server is responsible for making sure that damage does not propagate beyond what the client who made the link could do at the time that this client made the link, by using the open node that this client needed on the secure server in order to make the link. Thus if the link ACLs deny permissions to users coming into the secure server, the secure server must see to it that those permissions are always denied.

6 Final Remarks

The WBW storage system as we have defined it is just a basic system for storing information. Nothing is said about how to use this system.

We expect usage to be quite complex. For example, we believe a system can be developed that combines what could be done by a parallel processing shell, a UNIX `make` program, and a source code control system. But doing this, and then setting a standard for how it is to be done, is a very complex task.

Because the WBW storage system we have described limits itself to simply trying to store information and provide security for that information, the WBW storage system can be rigorously designed without a lot of experimentation. But systems that use the WBW storage system are real-world data bases and will only be developed with the aid of considerable experimentation.

7 To Be Done

When merging a write only object the client should be able to read a version of the object that has less info than the whole object but allows the user to write additions. E.g., a write only list should be implementable.

References

[GLO] The Globe distributed operating system. `http://www.cs.vu.nl/~steen/globe/`.

[LEG] The Legion distributed operating system. `http://legion.viginia.edu`.

[MF99] Gary McGraw and Edward W. Felten. *Securing Java.* Wiley, 1999.