DVSS: A Distributed Version Storage Server for CAD Applications

Denise J. Ecklund, Earl F. Ecklund, Jr., Robert O. Eifrig, and Fred M. Tonge

Computer Research Lab, Tektronix Laboratories
Tektronix, Inc., Beaverton, Oregon 97077 USA

Abstract

The Distributed Version Storage Server (DVSS) provides an underlying storage mechanism for a CAD-oriented data model. DVSS supports such project management features as version histories, alternate data versions, and multi-reader multi-writer access control in a heterogeneous network of workstations and fileservers.

Each design object is managed as a rooted directed acyclic graph (DAG) of versions. At any time, one path in an object's DAG is designated as its principal path; the current version in the principal path is the current version of the object. Other paths contain alternate versions of the object. Updates to any version path must be serializable, but derivation of alternate versions is not subject to this constraint.

Clients interact with DVSS using the *checkout/checkin* paradigm. Each object has a primary site, which synchronizes actions on the object. Group operations requiring multiple locks follow a deadlock avoidance scheme.

DVSS is robust in that it supports multi-reader and multi-writer data access in the presence of failures. Traditional data replication supports continued read access. Write-write conflicts resulting from continued write access during network partition are resolved at recovery time by creation of alternate versions. The cost of resolution is minimized by employing a unilaterally computable algorithm at each site.

1. Introduction

That CAD database applications have several properties not well supported by a traditional relational DBMS is a widely accepted view [1,3,4,5,7,12,13,14,16,17,19,20].

In particular, a database system for CAD must provide support for the following:

1. Long transactions that may extend over hours or days.
2. Complex relationships among components of a design.
3. Modeling designs as complex objects.
4. Design versions and alternative designs.
5. Modeling configurations of a design as complex objects.
6. Design data availability regardless of other users activities or system failures.

Further, it is frequently observed that design data objects have arbitrary formats. That is, the set of objects occurring in a design with the same format tends to be small, and is frequently a singleton set. This implies that database storage techniques based on multiple occurrences of objects with a regular format (e.g., records or tuples) are inadequate to manage design data.

In the Computer Research Laboratory of Tektronix Labs we are developing an experimental engineering design database system to be used by computer aided design tools. The system will support team engineering in a heterogeneous environment of engineering workstations inter-connected by a network. The network may also connect the workstations to one or more sites acting as file servers. Our development is being done on 68010-based UNIX[1] workstations connected with VAX[2] 11/780's.

It is our thesis that an engineering database system should have two levels: an abstraction management level and a storage management level. An example of a multi-layer computer-aided engineering system is shown in Figure 1. Versions and configurations of design objects are modeled at the abstraction level along with relationships and other semantic properties of design objects. The storage level provides primitive support for versions, configurations, and complex objects, including clustering and recursive retrieval.

In this paper we present DVSS, a Distributed Version Storage Server, as the storage manager in our experimental engineering design database system. We first describe our design goals and discuss related work. We then present the client's view of DVSS, followed by discussion of some aspects of the DVSS implementation. We close with a sketch of future work and conclusions.

[1] UNIX is a registered trademark of AT&T Bell Laboratories.

[2] VAX is a trademark of Digital Equipment Corporation.

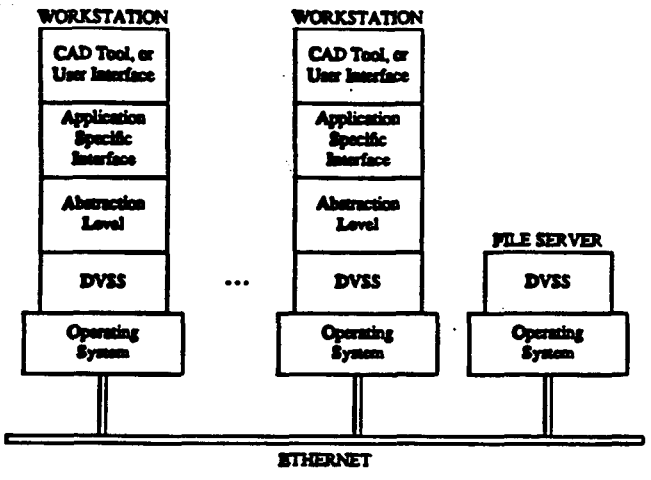


Figure 1: Layered CAE System

2. Goals

In the development of DVSS we set and attained four goals. Two goals (1 & 2) assert DVSS semantics visible to the client programs (or users), and two goals (3 & 4) address implementation issues.

1. Provide partition transparent storage system behavior.
2. Support distributed multi-reader and multi-writer access.
3. Provide a robust storage mechanism.
4. Minimize multi-phase protocols by using unilaterally computable algorithms.

Our primary goal is to provide a distributed system in which the behavior perceived by a user is independent of the current site configuration or connectivity of the network, in so far as possible. In particular the functioning of DVSS, while the system configuration is in a partitioned state, should not depend on the presence of any specific site or quorum of sites. The presentation of partition transparency is facilitated by using a multi-writer access paradigm.

DVSS' *multi-reader* and *multi-writer* access control paradigm was introduced by Ecklund and Price [10]. The multi-reader paradigm allows any number of clients to concurrently read any accessible copy of the same data. The multi-writer paradigm allows any number of clients to write new versions as successors of the same data. Under the DVSS multi-reader and multi-writer mechanism, requests that form read-write or write-read conflicts are not aborted, and write-write conflicts are resolved by the creation of alternate versions. Timing aspects of non-blocking reads and creation of alternate versions due to write-write conflicts can be visible to DVSS client programs and end users.

Partition transparent behavior is possible when reading data if any replicated copy of the data is accessible. Given that write-write conflicts are resolved by the creation of alternate versions, it is possible to allow all write operations to be performed locally while sites are separated by a communication failure. Thus, clients of the storage system may experience no disruption in service due to site or communication failures (depending on the placement of copies of data).

A robust storage mechanism provides reliable storage of data and maximal access to that data during failures. Our goal for robustness is to increase the likelihood of an object being available in a network partition, to allow a client to read any available copy of an object, and to allow all writes to complete. DVSS achieves robustness through reliable distributed protocols, data replication, and the multi-reader and multi-writer access control paradigm.

In any distributed system it is desirable to minimize the number of messages required to maintain mutual consistency among a set of sites. Most distributed protocols require multiple phases (e.g., 2 phase commit). In DVSS the number of message phases required by distributed recovery is minimized by defining unilaterally computable resolution protocols. In a *unilateral computation*, a set of sites performs a single exchange of state information. Upon receiving state information from the other sites, each site independently computes a mutually consistent result based on the information exchanged. No further negotiation phases are required.

3. Relation to Other Work

Several version servers have been proposed in the literature. These servers support intricate models of versions where many attributes and relationships such as time of creation, version type, access permissions, ownership, test status, type of database the version resides in, predecessor, successor, configuration, and equivalence are maintained by the server. Our storage server provides primitive support for those attributes and relationships that are essential in modeling versions. The abstraction level of an engineering database system must model arbitrary relationships and an arbitrary number of attributes. It is our contention that it is not desirable to model at the storage level any relationships that can be easily managed by the general mechanisms already required in the abstraction layer or by some minimal interaction between the two levels. Following is a brief discussion of three such version servers.

Katz, Anwarrudin, and Chang have proposed a distributed version server for CAE applications [15]. Their server supports multi-reader and single-writer access to untyped data. Structural relationships among the data versions are tracked by the server. Data is classified in three planes: the version plane relates temporal and alternate versions (which they call derivatives and alternatives respectively); the configuration plane relates those instances that comprise a version of a higher level object; and the equivalence plane relates different physical representations of the same entity. DVSS manages the version plane directly. The configuration plane and the equivalence plane are supported by DVSS group operations, but creation of versions in these planes is the responsibility of the abstraction layer.

Weiss, Rotzell, Rhyne, and Goldfein have proposed the Design Objects' Storage System (DOSS) [22]. DOSS stores temporal and alternate versions of objects and supports multi-

reader and single-writer access to those objects in contrast to DVSS's multi-writer access. The system is distributed in that any site having sufficient storage space stores copies of the data and functions as a server. All server sites maintain fully redundant directories of the data stored by the system. The non-server sites may cache information on the location of data objects they have referenced. The system provides for the permanent removal of machines from the network with facilities for migrating data stored on those machines to other sites, similar to DVSS's abandon and withdraw facilities.

Chou and Kim have proposed a system for controlling versions in a CAD environment [6]. The environment defines three database types: the public database, project databases, and private databases. Versions of CAD objects are classified as released versions, working versions, or transient versions. A version's classification restricts the operations that may be performed on that version and the type of database the version may reside in. DVSS provides a general federation mechanism that can be used by the abstraction layer to define databases with any desired scope of access. Chou and Kim's version server uses checkout, checkin, derivation, and promotion to achieve a multi-reader and multi-writer access control mechanism. Other major features of this model provide direct support for configuration management.

DVSS is unique in that it provides partition transparency and a federation mechanism to define arbitrary data access scopes. DVSS also supports multi-writer access, which has recently been incorporated by Chou and Kim. Of those proposed version servers that have been implemented, most provide distributed access to centralized objects. DVSS has been designed to provide robust distributed access to distributed objects.

4. Client View of the Distributed Version Storage Server

Clients of DVSS (e.g., a DML interpreter) implement a data model and its abstractions. These client programs view DVSS as a reliable distributed mechanism for storing and retrieving design data for teams of engineers. DVSS organizes design data under an association abstraction called a *federation* [9]. We have chosen the word federation to indicate that it is a loosely associated collection of sites and users whose participation in the federation is strictly voluntary. Formally, a federation is characterized by a triple {O, S, U}, where O is the set of data objects, S is the set of sites which belong to the federation, and U is the set of users who belong to the federation.

We anticipate that federations will be used to represent design projects, component libraries, software module libraries, public databases, and private databases. One user employs the *define* operation to create and name a federation. The defining user and the site local to the define operation are members of the federation. Other team members use the *enroll* operation to add their sites to the federations. The enrolling site must specify the name of the federation and the name of a site that participates in the federation. In this case, the distributed nature of the federation is not completely transparent to the users.

4.1 Users

Each user in a federation is a *virtual user* representing a set of related login accounts (e.g., those of a single user on different sites). Virtual users own and have access rights to the data objects stored by the DVSS. A virtual user is a capability consisting of a globally unique name and a password. A valid virtual user name and password must be provided, via the *openfed* operation, before any data manipulating operation can be performed in that federation. The virtual user mechanism may also be used to form groups of related users (e.g., the MMU design team) which are composed of distinct individuals with identical access rights to data objects in DVSS.

Virtual users are created by the *newuser* operation, and are destroyed by the *secede* operation (see figure 4). Each Virtual User is either an *associate member* or a *participating member* of a federation. Participating members may create and update objects. Associate members may only view the objects stored by the federation. The *moduser* operation allows a virtual user to change his name, password or promote himself from associate to participant status.

4.2 Objects

Clients store and retrieve objects using the *checkout/checkin* paradigm. A client process checks out a copy of an object, modifies the copy of the object, and *updates* the modified object by adding a new version of the object to the database. The storage server tracks the evolution of each object by storing and maintaining access to versions of each object. Objects themselves are not updated in place, but the directories, which contain information about the objects, are updated in place.

Versions of a single object are related by incremental refinement, by derivation, or by consolidation. Incremental refinement (see figure 2A) is a one-to-one relationship resulting from single-writer activity. Derivation (see figure 2B) is a one-to-many relationship between a version and a set of successor versions. Consolidation (see figure 2C) is a many-to-one relationship between a set of predecessor versions and a single successor version. A set of versions related by incremental refinement, derivation, and consolidation form a rooted directed acyclic graph of versions for each object.

When multiple users checkout the same version V of an object, the first user to perform an update operation will create a new successor version N, which is related to V by incremental refinement. Subsequent updates based on version V will create alternate versions of V. These alternate versions are successors related to V by derivation.

Each version has at most one successor related by incremental refinement; this is its *principal successor*. The principal successor is a member of the same version path as the predecessor, while each alternate version is a member of a distinct version path. Figure 2(B) illustrates versions with alternate successors related by derivation and a principal successor related by refinement. The derivation relationship may arise from multi-writer behavior.

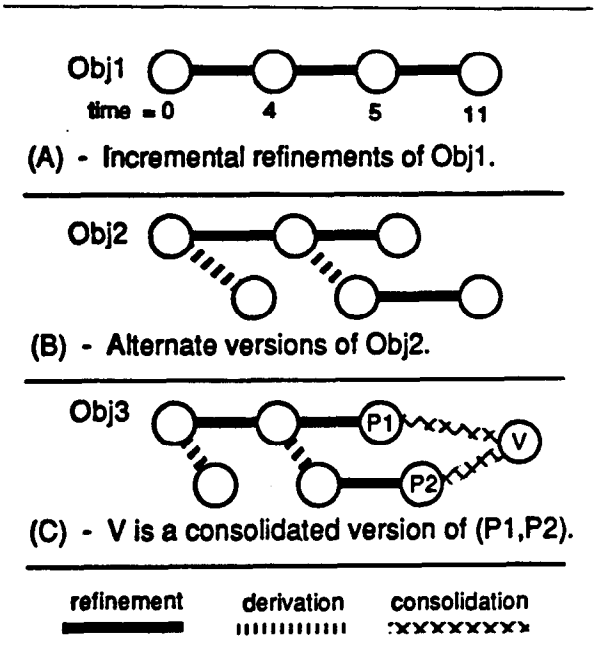


Figure 2: Versioning a conceptual object

When multiple writers have created a set of alternate versions $P_1, P_2, ..., P_k$, it may be desired that those versions be consolidated to form a single new version V. The version V is the source of a new version path in the version graph. Figure 2(C) illustrates a new version V formed from multiple predecessors $P_1, P_2, ..., P_k$. Version V has no predecessors related by incremental refinement, but a set of predecessors related by consolidation. The predecessor instances P_1 and P_2 may be updated by incremental refinement within their respective version paths.

4.3 Referencing Objects

Client programs use tokens to refer to subsets of related versions in a version graph. A token is a unique symbol #N assigned by DVSS. Figure 3 shows a DVSS object and its associated tokens. Each object has an associated token representing the graph of versions. Each version path in the graph has an associated token and an *implicit alias number*. A version path token or an object name with its version path implicit alias number may be used to refer to a set of versions related by incremental refinement. Each individual version has an associated token. Object tokens and version path tokens are used to make *floating references*. Version tokens make *fixed references*.

One version path in the graph is designated as the *principal version path*. The object token is a floating reference to the newest version in the principal version path of the version graph. In figure 3 version path Obj3(1) is the principal version path. The *assign* operation is used to select a version path as the principal version path of an object.

Any token that makes a floating reference may be suffixed with a time specification. The search for the referenced version begins by mapping the token to the current version in the specified version path. The incremental refinement relationship is used to traverse the path of incremental predecessors to locate the version that was current at the time specified. If the principal version path of an object O has been assigned to a different version path, then a time suffixed reference to O may map to different versions before and after the assign operation is performed. It is also possible that no version existed at the time specified.

When checking out objects, a client specifies a set of tokens. A floating token reference is mapped at checkout time to the appropriate version in some version path. A fixed token reference is mapped to a specific version, regardless of its relationship with other versions of the object.

4.4 Notification

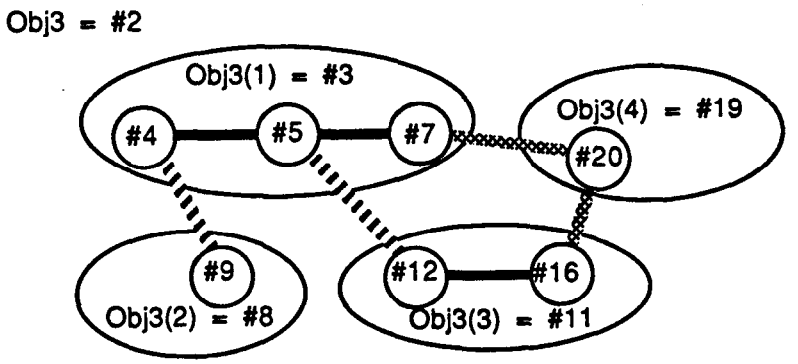
DVSS provides a notification mechanism whereby events known to DVSS and of potential interest to a user can be made known to that user. Examples of events which may be of interest to a user can include (1) conflicting checkouts (read-read conflicts), (2) conflicting updates (read-write and write-write conflicts), (3) resolution of conflicts and the unilateral creation of alternate version paths by the partition merge procedure, and (4) any state changes of DVSS directories.

Conflicting checkouts, conflicting updates, and the results of merge resolution are events that result in automatic notification of the users involved. Users may receive additional notifications by using the *ndefine* operation to specify events of which the user wishes to be notified. Events are specified by a pattern that defines a set of objects, a set of operations on those objects, and a set of users who will perform those operations. (A valid specification may include all objects in a federation, all operations supported by DVSS, or all users in a federation.)

DVSS persistently queues notification messages, describing the occurrence of a default or user specified event, until they are processed by the client who created the specification. The *nread* operation is used to read notification messages or, optionally, to block waiting for future notifications. Waiting for notifications does not block the entire client because the DVSS client interface provides many dynamically created concurrent sessions within a single IPC connection to the server.

5. Implementing DVSS

DVSS implements the services described above. Each site executes a single DVSS process serving all federations on that site. Each site stores, for each federation it is a member of, a fully redundant directory and partially redundant data. The directory at a site contains the full description of each object in the federations of which that site is a member. An object description consists of: the object's name; references to the object's principal path, the current version of the object, the virtual user who owns the object, the virtual users who may access the object; a full description of each version path in the object, a list of site locations for each version of the object, and partition tags used to maintain mutual consistency (see §5.3). We now



Name	Collection Type	Reference Type	Collection Token	Current Version
Obj3	Object	Floating	#2	#7
Obj3(1)	Version Path	Floating	#3	#7
Obj3(2)	Version Path	Floating	#8	#9
Obj3(3)	Version Path	Floating	#11	#16
Obj3(4)	Version Path	Floating	#19	#20
Obj3(1)[$time_0$]	Version	Fixed	#3	#4
.	.	.	.	.
#16	Version	Fixed	#11	#16
.	.	.	.	.

Figure 3: A DVSS Object and its Associated Tokens

discuss in more detail how our DVSS implementation stores design objects, performs operations on the relationships among versions, and makes the system robust in the event of site or communication failure.

5.1 Storing Objects

DVSS stores multiple copies of each object version and manages the relationships of incremental refinement, derivation, and consolidation among those versions. Storing redundant copies of each version provides increased speed of access and increased availability of data. Default replication factors can be set for each object or on a federation-wide basis. The replication factor and the most-preferred storage sites can be specified by the user at the time of a version's creation (in the *create*, *update*, *uphold*, or *write* operations). If such a specification is not made, the replication factor and preferred site list are inherited from the object. If the object has no such specifications, federation wide defaults are used.

DVSS attempts to conserve storage space by storing related object versions as differences. If copies of a version and its immediate predecessor(s) both reside on the same site, DVSS will store the predecessor(s) as a backwards difference [21] with respect to the given version. If a predecessor has more than one successor, an arbitrary choice is made of the successor to difference from. Because floating references should be more common than fixed references to older versions, the newest version in each version path is stored in its entirety. (The practice of storing backward differences of old versions related by incremental refinement has been shown [18] to reduce the storage space consumed by the old versions by as much as 98 per cent.)

DVSS attempts to maximize the benefits of backward differences by storing replicated versions on the same sites as their predecessors whenever possible. Calculation of a difference can be time consuming and for this reason is performed as a background process.

	Federation Management Operations
Abandon	removes an inaccessible site from a federation without removing data from the site.
Define	creates a new federation (which exists only on the client's site).
Enroll	add a new site (the site where the enroll is executed) to a federation which exists on a remote site.
Moduser	allows a Virtual User to change his name, password and/or convert from associate to participant status.
Newuser	creates a new virtual user.
Secede	allows a virtual user to remove himself from a federation. Any references to the seceding user are optionally replaced with references to a virtual user whose name and password are specified (otherwise all references are inherited by the definer).
Withdraw	removes a site from the federation after salvaging the data stored on that site.
	Group Operations
Checkout	reads a set of object versions (for update).
Read	reads a set of object versions (not updateable).
Return	discards DVSS' record of a checkout without updating any of the objects.
Update	adds new versions to checked out objects. The set of versions can be a subset of the versions checked out, with the remaining versions being returned.
Uphold	adds new versions to checked out objects and creates a new checkout of the versions currently held by the client.
Write	adds new successor versions to a set of objects without previously checking out their predecessors.
	Single Entity Operations
Assign	changes the primary path of a data object.
Create	creates an initial version of a new object in a federation.
Delete	logically removes an object from a federation. Delete fails if versions are checked out.
Derive	creates an alternate version path as the successor of a version.
Erase	updates a version with a copy of its predecessor or logically removes an entire version path from an object. If any of the erased versions have been checked out, an alternate path will be derived when the corresponding update occurs.
Newown	allows the owner of a version path or object to give away the ownership of the version path or object along with all of its privledges.
Setperm	sets global permissions and access lists for each object and version path.
Usurp	allows the owner of an object to acquire the ownership of a version path of that object.
	Miscellaneous Operations
Closefed	terminates a session within a client-to-DVSS connection.
Copysites	changes the default replication and/or copysite locations of a federation or object. When applied to a specific version, the current locations and replication factor are changed.
Enquire	provides rudimentary query facilities for DVSS directory information.
Nactive	enables (or disables) a message queue for receiving new notification messages.
Ndefine	creates a notification message queue and an event selector (pattern).
Ndestroy	destroys a notification queue and any queued messages.
Nread	reads notification messages from a named queue or waits for new notifications.
Openfed	creates or re-defines a concurrent session within a single DVSS connection. The session may optionally restricted to a single federation or a single virtual user.

Figure 4: DVSS Operations

5.2 Operations

The DVSS client interface consists of operations such as *assign*, *checkout*, *create*, *define*, *delete*, *enroll*, *return*, *setperm*, and *update*. Figure 4 gives a brief description of the set of DVSS operations implemented at this time.

The checkout, read, return, update, uphold, and write operations are *group operations* because they can deal with a set of versions in an atomic manner. This mechanism is designed to support operations on complex entities (i.e., entities represented by many DVSS objects). DVSS does not currently provide a means for constructing the group of objects from object references stored in other objects (e.g., configuration objects). A recursive checkout mechanism will be implemented at the storage level during integration with the abstraction level (see §6.2).

The checkout operation reads a set of object versions with intention to update; the read operation reads a set of object versions for viewing only. The return operation returns checked out versions without update. The update operation adds new versions to some or all of a set of checked out objects; objects not modified are returned. The uphold operation performs an update of modified versions followed by a checkout of the total set of versions now held by the client. The write operation adds new versions to a set of objects without requiring a previous checkout.

The group operations of read and checkout can span multiple federations. The group operations of update, uphold, and write may be executed in one federation only. Given that a federation is used to control the scope of access, an individual engineer may wish to read or checkout design data from other federations (e.g., a library), but it is not reasonable to update designs from distinct federations in a single update operation. Instead, one would expect to create local objects in the current federation for the modified library objects.

In an environment free of communication failures, the directories on all sites in a federation maintain mutual consistency under the following algorithm. An operation is initiated by client software on site A and communicated to DVSS by interprocess communication. DVSS acquires all appropriate locks on the directory entries to be modified by the operation. Each partition (see §5.3) has one lock site. A single lock request message acquires all locks (in an order which precludes deadlock). Once all locks have been acquired, site A performs and commits the modifications to its local directory structures. Shadowing of the directories is used to ensure atomicity of each DVSS transaction. Next, a multicast message is sent to all other sites in the federation. The message informs these sites to update their directories in an identical manner. Once all sites have acknowledged the directory change, site A releases its locks.

Note that only the directory entries are locked and updated in place. New copies of the data, which constitute versions of the objects themselves, are moved between DVSS and its client before the directories are locked and modified, and between sites (if necessary) after the directories are modified and unlocked.

5.3 Achieving Partition Transparency

The multi-reader and multi-writer paradigm allows DVSS' robustness mechanism to hide almost all site and communication failures. Multi-readers can continue as long as replicated object versions are reachable. Multi-writers may always continue. Conflicting updates from non-communicating sites are detected when the communication failures are corrected.

To achieve failure transparency, DVSS must distinguish between failures and the correction of failures. The fault-tolerant *Virtual Partition* control algorithm [11] is used as a basis for tracking sets of communicating sites. A virtual partition is a named collection of sites that believe they can communicate with one another. Each partition name is a unique token from an ordered set. Every site maintains a history of the names of its virtual partitions. When DVSS processes client requests, each site will communicate only with the sites belonging to its current virtual partition. When a directory entry is modified, the name of the current virtual partition is written with the modified data. This partition tag obviates the need for a separate log of changes made within a partition.

A possible site or communication failure within a virtual partition is detected when one or more sites do not receive a reply from another site in a timely manner. The correction of a possible failure is detected when one site receives a message from a site outside its current virtual partition. (Each site utilizes a *finder task* that periodically attempts to send messages to sites outside of the local site's current virtual partition.) When a possible failure or correction of a possible failure is detected, recovery begins with a two-phase protocol to form a new virtual partition. In phase one, one or more sites send proposal messages to all known sites. A site receiving a proposal may accept the invitation if the partition name proposed in the invitation is "larger" than any partition known to this site. An accepting site acknowledges the invitation by sending its partition history to the initiating site. If the proposed partition name is not acceptable, then the receiving site may become an initiator and send partition proposal messages to all sites. If an initiator has received no "better" invitations and has received or timed out all acknowledgements, a commit message is sent to all sites that acknowledged the invitation. The commit message specifies the list of sites that will form the new virtual partition. For each virtual partition there exists a distinguished site called the *partition initiator*, that is, the site that successfully proposed the newly formed partition. Client requests in the recovering federation are suspended during the recovery process.

When a site or communication failure occurs, a set of surviving sites (partition) S loses contact with the set of down or disconnected sites D. The new virtual partition S performs a *divergence recovery* to achieve a mutually consistent view of the objects and sites in S, and must continue to process requests for their local clients. Mutual consistency is achieved by propagating any multicast messages originating from a site in D and not received by a subset of the sites in S. Continued client service is achieved by releasing all locks that are held by requests originating at sites in D. If the lock site for partition $(S \cup D)$ is a member of D, the partition initiator is selected as the lock site for partition S.

When a failure is corrected, two or more partitions will merge to form a single mutually consistent partition. *Merge recovery* achieves mutual consistency by decommissioning the lock sites of the merging partitions, by selecting the partition initiator as the new lock site, and by propagating among the set of merging sites all results that are unknown by any of the merging sites.

Merge recovery begins with the two-phase negotiation protocol. The commit message specifies the list of sites that will form the new virtual partition, a *representative site* for each of the merging partitions, and the partition history for each of the merging partitions. By comparing the merging partitions' histories, each representative site calculates a mutually exclusive set of partition names that are part of the representative's history and are omitted from the history of at least one of the merging partitions. Based on this mutually exclusive set of partition names, each representative constructs a *change list* containing all directory information whose associated partition tag value is in this set.

In the third phase of a merge recovery, all sites acquire a copy of each change list. Each change list consists of two distinct parts: (1) information about each group update (write or uphold) operation, ordered by update tokens; and (2) information on modified directory entries, ordered first by object token, next by version path token within the object, and finally by version token within the version path.

Change lists distribute new information and enable the discovery of conflicting requests. The process of conflict resolution is defined so that each merging site unilaterally computes the same resolution. To resolve conflicting group updates, each merging site reads and stores all group update information from the first part of each change list. From this information the group update conflicts are easily detected. When a set of conflicting group updates are discovered, at most one update may be selected as the winning update (i.e., the versions added by that update remain in the version path they were originally placed in) and all losing updates will eventually be added to the version graph as alternate versions. The algorithm for resolving conflicts attempts to maximize the number of winning group updates. Finding the maximal solution is an NP-Complete problem. DVSS employs heuristics to order all group updates reported by the merging partitions. A greedy algorithm is then applied to select an approximately maximal set of non-conflicting group updates.

The heuristic used by DVSS to rank each group update is a function of the *breadth* and *depth* of the update. The breadth of a group update is the number of new versions added by that update. The breadth of an update is used as an indicator of the complexity of the entity being updated. The more complex the entity, the more complex the update activity; therefore, we would like the most complex updates to be winning updates. The depth of a group update is the maximum depth of any version added by that update. The depth of a version is the number of predecessor versions that are reported in the same change list. The depth of a group update reflects the amount of update activity on specific version paths. Note that depth may be an inadequate measure of the complexity of the update activity because some DVSS clients will perform updates (creating new versions) more often than other clients.

Selection of winning group updates begins with the list of all reported group updates (ordered by the heuristic weighting), an empty set of winning updates, and an empty set of losing updates. Each update in the ordered list is tested for conflict with any update in the set of winning updates. The conflict test is stated as follows: A group update G conflicts with a winning update if (1) G updates any version path updated by any winning update, or (2) for each update P that created a predecessor version of one of the versions created by G, the update P conflicts with any winning update. Clearly, the second condition of the conflict test is recursive. If an update G does not conflict, then G and all updates that created a predecessor of one of the versions created by G are removed from the ordered list of updates and added to the set of winning updates. If an update G conflicts, then G and every update that created a successor of one of the versions created by G is removed from the ordered list of updates and added to the set of losing updates.

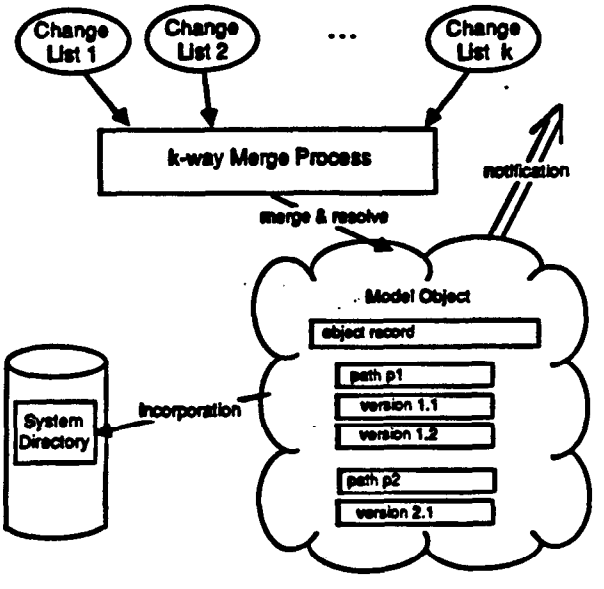


Figure 5: Processing Change Lists for a K Partition Merge.

When all winning group updates have been determined, each merging site reads and processes the second part of every change list. Given that each change list is ordered by tokens, the merging sites process the information in a manner akin to a K-way merge. Figure 5 illustrates the basic merge strategy. For each object token appearing in one or more change lists, a model of that object's DVSS directory entries is created in memory using information from the change lists, the selected set of winning group updates, and the local disk resident directory. If multiple change lists report updates to the same version path, the set of winning updates (computed from part one information) specifies, as the true continuation of the version path, a set of versions from at most one change list. All versions resulting from losing updates are added to the object model as

new alternate version paths. New version path tokens are unilaterally created for these paths. When the model is completed, it is *incorporated* into the local system directory. Incorporation is the act of adding all new information and over-writing all modified information.

If a site fails to complete a merge recovery (because the site fails or a representative site fails), that site must roll-back to its previous partition state and attempt to initiate a new virtual partition. If a site successfully completes the merge process, that site's partition history is the union of all of the merging partitions' histories. Complete details of divergence recovery, merge recovery, conflict resolution, and incorporation are contained in [8] along with arguments that divergence recovery and merge recovery maintain mutual consistency within a newly formed virtual partition.

5.4 Replication and Migration

DVSS keeps a vector of sites ordered by storage priority for every version of every object. The vector is distributed to all sites in the current partition as part of the version creation multicast message. The vector may specify sites that are not in the current virtual partition. Once a directory entry for the new version has been created, background tasks are created on the first m multicast receivers (in the current partition) listed in the vector to acquire copies of the new version from the creating client's site. m is the minimum of the replication factor (n) and the number of multicast receiving sites in the current partition. If a site's attempt to acquire a copy is successful, that site will multicast this fact to the current partition so that directories can be updated.

Note that the creating client's site will always appear in the copy site vector, but it may not be among the first n sites. In this case, the number of copies can become $n+1$ if all sites acquire copies. When the number of copies exceeds n, the least-preferred sites (in this case, the creating client's site) can remove their copies of this version. This removal is done conservatively by requiring that each storage site confirm the presence of a copy prior to removing the local copy.

The *copysites* command can change the replication factor and/or preferred site list for a federation or object; this modifies the defaults used for future version creation. When the *copysites* operation is applied to a specific version of an object, the sites on which that version resides are changed. A new vector of sites is either specified explicitly or generated from the defaults. Any sites that currently hold copies and are not listed in the new vector are added to the new vector as the least preferred sites. When a site discovers that it is one of the m most favored sites and it does not hold a copy, it will attempt to acquire one.

The procedure described above is called *propagation* because it is performed on a version by version basis, to be distinguished from *migration* which deals with moving versions between sites due to changes in the current partition or due to the *withdrawal* of a site from the federation. Each server runs one background migration task that examines all object versions to see if the local site needs to acquire or remove a local copy. The mechanism is the same as described for propagation. Migration is also used on sites with archival storage (e.g., optical disks) to absorb obsolete object versions. In this case, the multicast of acquisition by an archival site will enable all non-archival sites to remove their copies.

6. Ongoing Development

The first prototype of DVSS was completed in 1985, and the present (rearchitected) prototype has been running since 1986. Our early emphasis was on the functionality of the distributed multi-reader/multi-write algorithms and the recovery mechanism. We are now enhancing DVSS to meet the client interface requirements of a data model. In the following sections we discuss issues associated with three of our current efforts.

6.1 Storage Reclamation

Storage reclamation is required for client operations *delete* and *erase*, which explicitly deallocate versions and their associated directory entries. In addition there are entities in the DVSS' directories generated by client operations that can be re-used. Examples include (1) access list slots freed by the *setperm* operation, (2) the list of checked out versions freed by *update*, (3) the virtual partition history list for a site, and (4) notifications already read by a client. We believe that tagging data as "removed" without reclaiming the reusable storage (*logical removal*) is not adequate in the long term. However, the logically removed state is necessary in the short term to provide partition transparency. A logically removed entity is *reclaimable* on a site if it is known on this site that all sites in the federation have knowledge of its logical removal. Knowledge of logical removal of an entity is inferred from the confirmed site membership lists for each partition in the partition history. These partition histories are communicated to the initiator site who then determines (by transitivity of the predecessor partition relation) the set of partitions that are reclaimable. Logical deletions, like any other changes in the DVSS, are tagged with the partition in which they occur. If a logical deletion is tagged as occurring in a reclaimable partition, it can be *realized*. We note that (1) the predecessors of reclaimable partitions are reclaimable and (2) a partition containing all sites in a federation (i.e., a *maximal partition*) is always reclaimable. The partition history is truncated at the most recent reclaimable partition along each directed arc in the partition history digraph.

DVSS servers will attempt to perform storage reclamation under either of the following situations: (1) following a merge recovery, when it becomes known that all sites successfully completed the merge processing, then new reclaimable partitions are calculated and directory space freed during those partitions is realized; (2) while operating in a maximal partition, after a sufficiently large number of logical deletions have taken place, a two phase synchronization protocol is carried out to ensure that the partition is still maximal and all logical deletions carried out during the maximal partition can be realized. If a site fails during reclamation, the reclamation transaction will be backed out when the site recovers, thus deferring that site's realization of the deletions until the next merge recovery.

6.2 Integration with Data Model Clients

While implementing DVSS, we have been developing a data model (TEDM) that fulfills the modeling requirements for an engineering database system (see Anderson, et al. [2]). We intend that an object-oriented data model, such as TEDM, be the client of DVSS. Although one might map TEDM objects to DVSS storage objects in a one-to-one manner, and deal with the efficiency considerations of providing storage and access to "small objects", we intend to treat DVSS storage objects as storage segments. The data model must cluster its objects into DVSS segments based on relationships existing among instances of the objects. (For design databases, clustering on relationship attributes is expected to have more impact than clustering on value attributes.)

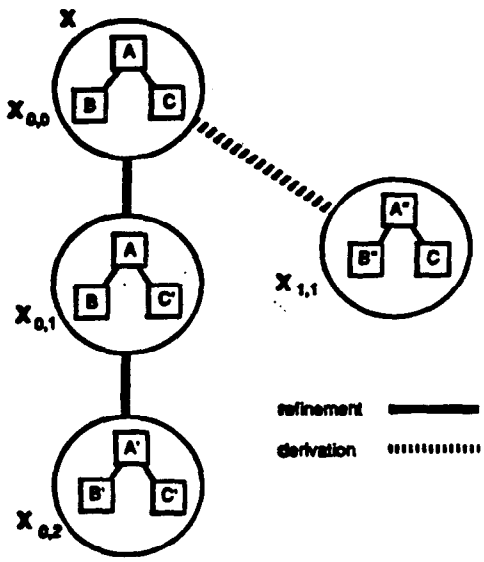


Figure 6: Versions of objects A, B, and C clustered in versions of segment X

Both the abstraction level (e.g., TEDM) and the storage level (i.e., DVSS) of our system have some notion of versions and their associated semantics. A straight-forward technique for integrating these systems would be to store a cluster of TEDM objects in one DVSS segment, and represent each version of the cluster in a version of the segment. Figure 6 illustrates versions of segment X containing versions of the clustered objects A, B, and C. Notice that a new version of C can be made without modifying A, but the same may not be true of B. Issues such as when a new version of a component triggers a new version of the containing complex object (i.e., version propagation [17], or percolation [3]), and when an update is an update-in-place or when it is versioned must be managed by an engineering DBMS. We are developing a specification for the semantics of versions to ensure that there are no incompatibilities between the client data model version semantics and DVSS version semantics.

A significant task in integrating TEDM and DVSS is making the meta-information regarding the format of TEDM objects accessible to and interpretable by DVSS. This is necessary in order to encapsulate the architectural heterogeneity of the computing environment within DVSS. Inter-site access must take architectural disparities (e.g., byte ordering, floating point formats) into account where applicable; DVSS will send its objects in a network standard format, which can be configured to agree with the architecture of the workstations. Another need of DVSS for TEDM scheme information is to perform atomic recursive checkout of a configuration object. Access to the scheme of objects will also be required to use fields within objects as the units of granularity for DVSS' differencing algorithm.

In principle, DVSS will provide both transparent data distribution and concurrency control to its client data model. We intend to integrate a single-site single-user implementation of TEDM with DVSS, leveraging off DVSS for both its multi-user and distributed data capabilities.

6.3 Productivity under a Multi-Reader and Multi-Writer Mechanism

The multi-reader and multi-writer access control mechanism supported by DVSS is an optimistic mechanism in the strongest sense. Compared with single-writer mechanisms, the clients of DVSS achieve higher throughput, which affects productivity of the end-users (e.g., design engineers). Under a multi-reader and multi-writer access control mechanism, clients can create alternate versions unintentionally. In such cases, it is likely that consolidation will be performed to effect a combined version. The end-user cost of performing consolidation must be weighed against the increased throughput to determine whether or not a multi-reader and multi-writer access control mechanism increases the productivity of the system's end-users. We are currently building simulations of a single-writer distributed system based on locking and of DVSS. We intend to compare the two systems under several models of user behavior in an attempt to measure the effects of each access control mechanism on end-user productivity.

7. Conclusions

We have developed a Distributed Version Storage Server for the storage level of an Engineering Design Database Management System. DVSS is intended to be integrated with a data model client, which will provide a suitable abstraction level. Designing DVSS has presented two interesting problems: developing the semantics of a versioned storage model, and developing and implementing distributed algorithms for the DVSS operations. The latter was by far the more challenging.

DVSS is designed to be robust and in so far as possible to make failures transparent to its clients and users. We wish to support ongoing use of the system by clients during network partitioning in such a way that the system behaves consistently whether or not a failure has occurred. Supporting the ability of users in more than one network partition to concurrently effect operations against DVSS precludes using a concurrency control mechanism based on a voting algorithm. Fortunately, versions and multi-writer algorithms for concurrent operations can be

used to provide service during network failures in a manner that appears uniform to the client, except for delayed notification due to concurrent operations on the same objects.

In a distributed database system where conflicting computations can proceed concurrently in two or more partitions, the amount of data that must be exchanged to process merge recovery can be substantial. In addition, conflicts must be detected and resolved as there is no guarantee that all actions are serializable. Thus, the recovery protocol must guarantee that a consistent resolution of conflicts occurs at each site. The agreement on a new virtual partition and exchange of recovery merge data requires a two phase protocol. Verification of a consistent merge resolution would require an additional protocol of at least two phases. A merge algorithm that performs a unilateral computation at each site and that produces a provably consistent result of the merge avoids this costly verification protocol.

Acknowledgement

We thank the VLDB reviewers for their helpful comments and suggestions on improving this paper.

References

1. *The Department for Defense Requirements for Engineering Information Systems, Volume 1: Operational Concepts.* ed. J. L. Linn and R. I. Winner, July 2, 1986.

2. T. Anderson, E. Ecklund, and D. Maier, "PROTEUS: Objectifying the DBMS User Interface," in *1986 International Workshop on Object-Oriented Database Systems*, pp. 133-145, September 1986.

3. T. Atwood, "An Object-Oriented DBMS for Design Support Applications," in *Proceedings of Computer Aided Technologies COMPINT 85*, IEEE, Montreal, Quebec, Canada, September 1985.

4. F. Bancilhon, W. Kim, and H. Korth, "A Model for CAD Transactions," in *Proceedings of the Eleventh International Conference on Very Large Data Bases*, pp. 25-33, Stockholm, Sweden, August 1985.

5. J. Banerjee, W. Kim, D. Woelk, N. Ballou, and H. Chou, "Database Support for Object-Oriented Applications (Extended Abstract)," in *Workshop on Information System Support for Integrated Design and Manufacturing Processes*, 1986.

6. H. Chou and W. Kim, "A Unifying Framework for Version Control in a CAD Environment," in *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pp. 336-344, Kyoto, Japan, August 1986.

7. H. C. Du and S. Ghanta, "A Framework for Efficient IC/VLSI CAD Databases," in *Proceedings of the Third International Conference on Data Engineering*, pp. 619-625, February, 1987.

8. D. Ecklund, "Robustness in a Distributed Storage Server for Engineering Design Data with Versions," *Ph.D. Thesis, Oregon State University*, December, 1986.

9. E. Ecklund, D. Price, R. Krull, and D. Ecklund, "Federations: Scheme Management in Locally Distributed Databases," in *Proceedings of the Nineteenth Hawaii International Conference on System Science*, pp. 395-407, 1986.

10. E. F. Ecklund, Jr. and D. M. Price, "Multiple Version Management of Hypothetical Databases," in *Proceedings of the Eighteenth Hawaii International Conference on System Sciences*, pp. 163-173, 1985.

11. A. ElAbbadi, D. Skeen, and F. Cristian, "An Efficient, Fault-Tolerant Protocol for Replicated Data Management," in *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 215-229, Portland, Oregon, March 25-27, 1985.

12. M. Gray, "Databases for Computer-Aided Design," in *Proceedings of the Second International Conference on Databases*, pp. 247-258, Heyden, September 1983.

13. C. Jullien, A. Leblond, and J. Lecourvoisier, "A Database Interface for an Integrated CAD System," in *Proceedings of the Twenty-third ACM/IEEE Design Automation Conference*, pp. 760-767, June 1986.

14. R. Katz, "Transaction Management in the Design Environment," in *Proceedings of the Second International Conference on Databases*, pp. 259-273, Heyden, September 1983.

15. R. Katz, M. Anwarrudin, and E. Chang, "A Version Server for Computer-Aided Design Data," in *Proceedings of the Twenty-third ACM/IEEE Design Automation Conference*, pp. 27-33, June 1986.

16. W. Kim, H. Chou, and J. Banerjee, "Operations and Implementation of Complex Objects," in *Proceedings of the Third International Conference on Data Engineering*, pp. 626-633, February, 1987.

17. G. S. Landis, "Design Evolution and History in an Object-Oriented CAD/CAM Database," in *Proceedings of Compcon Thirty-First IEEE Computer Society International Confe rence*, San Francisco, California, March 1986.

18. D. Leblang and G. McLean, Jr., "Configuration Management for Large-Scale Software Development Efforts," in *Workshop of Software Engineering Environments for Programming-in-the-Large*, pp. 122-127, Harwichport, Massachusetts, June, 1985.

19. R. Lorie and W. Plouffe, "Complex Objects and Their Use in Design Transactions," in *Proceedings of the 1983 ACM Engineering Design Applications*, pp. 115-121, May, 1983.

20. D. Price and D. Maier, "Data Model Requirements for Engineering Applications," in *Proceedings of the International Workshop on Expert Database Systems*, ed. L. Kerschberg, 1984.

21. D. Severance and G. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," *ACM Transactions on Database Systems*, vol. 1, no. 3, pp. 256-267, September 1976.

22. S. Weiss, K. Rotzell, T. Rhyne, and A. Goldfein, "DOSS: A Storage System for Design Data," in *Proceedings of the Twenty-third ACM/IEEE Design Automation Conference*, pp. 41-47, June 1986.