OpenSMS

Stephen Cranage
StorageTek
Steve_Cranage@stortek.com

Abstract

Systems Managed Storage is a proven concept in traditional mainframe computing. Client-Server operating systems have traditionally lacked the tape I/O subsystem, file system intelligence, and data classification policies required to implement the storage management services that are necessary attributes of a scalable data processing environment. OpenSMS is an Open-source framework that addresses these deficiencies.

1. Introduction

The vision of ILM, or Information Lifecycle Management, has become the topic of great industry attention of late. Rising complexity and cost of storage management has become increasingly apparent. The previous generation of mainframe processing professionals viewed storage management as a systemic task, tied into the base operating system. In the process of migrating applications to new client/server platforms, we have lost some essential operating system services in this area.

While taking advantage of lower cost distributed computers, we have migrated systems that were traditionally departmental and desktop platforms into enterprise computing, and lost sight of how data growth would pose challenges when it occurred on operating systems that were not designed to manage enterprise storage. Situational awareness is now creeping up on the industry as the realization is setting in that scalability and ad-hoc management of storage are mutually exclusive concepts.

Backup processes are a good example. Michael Peterson, Program Director of SNIA's Data Management Forum reports that: [1]

A strong driver exists pushing the revolution to disk-based data protection and it is not cost. Cost is merely an enabler. The driver is IT's urgent need to solve the backup problem. IT has to reduce operating costs and cope with a smaller staff. IT must "stop backing up" to solve this complexity problem. It is the only real way. The solution requires a fundamental shift in architecture, moving to a simple, transparent operation where redundancy is native to the write process, where data is always there, and even the concept of "restore" goes away.

Making data protection native to the write process, and making "restore go away" is a fundamental function of some traditional Hierarchical Storage Management (HSM) systems, which typically act to duplicate a file object in seconds to minutes after creation or modification, and if well designed, integrate this process into a comprehensive data protection model. Many other aspects of the ILM vision can also be addressed by HSM concepts; these include dealing with regulatory compliance and archiving issues that are not well served by the current crop of fixed content products on the market today

These products use high density disk arrays for storage in spite of the fact that storage comprised of massive arrays of spinning disk spindles is a poor choice for long term data retention due to the perishable nature of the underlying disk technology, and the high cost of maintenance and power for these technologies. The high operating costs combined with rapid technological obsolescence drives the need for routine retirement and replacement of the technology. Unfortunately, increasing array density makes this very problematic. The issue is movement of massive amounts of data off an obsolete platform to the new, and the affect on application availability.

Again, HSM concepts can address the problem. Classification of file objects as they are created can direct the duplication of data to an appropriate tier, including a tier with good archival properties, or good quality of service properties, or both, based on policy directives. In fact, HSM systems are very common in the mainframe environments, where they are an integral component of a larger solution that is referred to as Systems Managed Storage.

Many HSM solutions have come to market for client/server platforms over the years. However none has achieved broad commercial success to the extent of becoming common in client/server environments, as they did in mainframe computing. There are a number of reasons for this, including poor product focus or reliability or poor hardware choices in implementation, but the principal reason is the technical challenge of operating system integration in a multi-platform, multi-operating system environment.

2. OpenSMS

One of the biggest challenges in creating a HSM product lies in being able to return migrated files back from multi-tier storage in a manner that is transparent to applications. Historically, creating data management applications that intervene within operating system services such as file I/O has been a difficult and expensive process given the proprietary nature of commercial operating systems. As evidence, note the lack of multi-platform products in the HSM space. Vendors who have entered this space have been compelled by intricate dependencies on proprietary kernel code to pick a single operating environment to support.

In the mid 90's, the industry developed an API for data management that was designed to address this set of problems. The Data Management Application Programming Interface (DMAPI), was designed as a standardized set of "hooks" into the file system that would allow data management companies to write HSM software to a standardized file system API [2].

DMAPI interfaces appeared in many file systems on virtually all computing platforms in the years since. Although compatibility between the interfaces is not perfect, dealing with the minor incompatibilities in DMAPI implementations is quite manageable (unlike maintaining compatibility with evolving proprietary operating system internals).

Silicon Graphics (SGI) recently released its high performance XFS file system for Linux under the GNU General Public License[3]. This gave the entire industry access to the source code for a DMAPI enabled file system. Shortly thereafter, IBM followed suit with its JFS file system[4], and based their DMAPI implementation on SGI's implementation.

We seized this opportunity to create a policy-based data mover framework. We called this framework OpenHSM, and published its source code under General Public License (GPL). In order to have something to move archival data to, we took an enterprise Tape Management System (TMS), which StorageTek previously sold as a commercial product called REELlibrarian, and published its source code under the GPL, we call that OpenTMS. Taken together we refer to the two projects as OpenSMS (Open Systems Managed Storage).

2.1.1. Architectural Approach

Our approach is differentiated from traditional HSM products in the way we view storage of file objects in the tape management system (TMS). We view the file system and TMS to be "parallel universes" of data storage. Each has properties that make it better for one storage requirement or another, but each is a storage namespace where file objects are directly accessible, regardless of where they reside.

While the concept of a TMS, or for that matter a tape I/O subsystem by itself, may not be intuitive to everyone, it is well tested in legacy mainframe environments, as well as in legacy UNIX based supercomputing environments.

In the last decade, when large monolithic high performance computers were more common than they are today, there were a few UNIX variants from Cray and others that did include tape I/O subsystems services. These services include device allocation, a mount request system and low level device control. The TMS services (often the very same REELlibrarian code licensed from StorageTek) provide an additional layer of intelligence that includes file cataloging, and media management.

While the TMS is a separate namespace that is in some ways comparable to the file system namespace, it also has some significant differences. The physical properties of sequential access media of course impose some limitations such as single user access and latency. But TMS services also have beneficial properties that make them an indispensable component in creating a scalable archive. Some of the distinguishing characteristics of the TMS namespace include:

- Enterprise wide scope. The TMS client services can satisfy access for a file object anywhere the TMS services are installed. This code is all user level, and has been historically ported very widely. Since TMS access methods don't include multi-user access, this is simpler than a shared file system. Shared SAN based tape transports can be used to distribute large amounts of data at high-sustained transfer rates through the TMS services using channel protocols for the transport without the complexity of a SAN file system.
- Logical vs. hierarchical organization. TMS file objects are stored in a logical container called a "volumeset". The volumeset has the form of userid/volumeset_name:Vno:Gno. Vno and Gno refer to version and generation numbers. Within each of these volumesets, is a flat namespace where files are stored and individually cataloged. We have individual

policy directives that organize data into volumesets logically, so for example when a user writes a file with a ".avi" extension to a managed file system, it is written to a volumset named "Movies", with that user's userid.

- Policy attribution. Volumesets are created with attributes that are used by the TMS to manage their life cycle automatically.
- Scalability. The TMS volumeset is essentially infinitely large in terms of raw storage capacity. It grows as long as data is written to it, and there are compatible scratch volumes available.

2.1.2. OpenSMS Storage Topologies

We view the storage market as having produced many highly granular types of storage systems with different performance and cost metrics, and wanted the OpenSMS framework to fully support data movement between an arbitrary number of storage tiers. We allow for each file system tier to be accessed independently of all other tiers. In complex multi-tier implementations, data faults cascade until they reach a tier where the data still resides. Policies copy data between tiers.

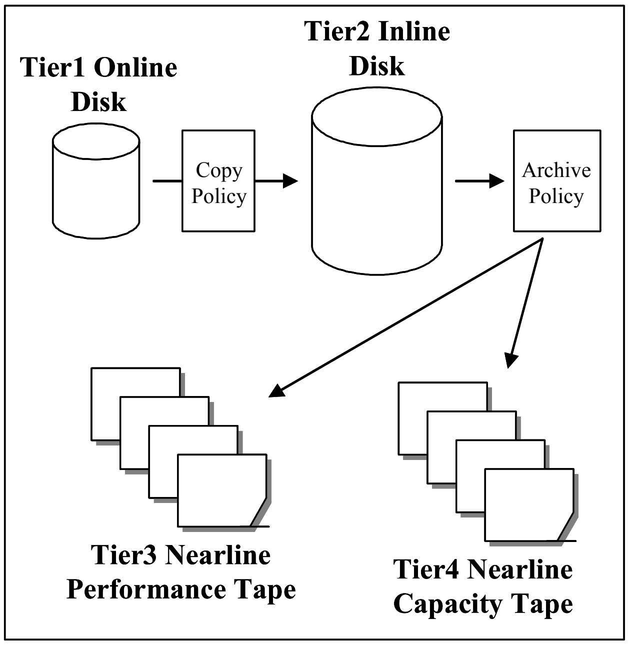

Figure 1. Multi-tier Storage Topology.

Conceptually, we divide our policies into two camps, copy policy (file system to file system) and archive policy (file system to TMS). Each has the ability to support block release, and application-transparent recall from its copy/archive target destination.

OpenSMS is designed so that these policies can be intermixed and combined in various storage topologies, including possibly quite complex topologies with many tiers of storage containers with different performance, reliability, and durability (archival) characteristics. Any arbitrary number of policies can be running against any file system, and the purpose of these policies is to either copy or archive a file system object sometime in the first few moments of a file's creation (or modification) to another storage container.

We think of this as near "real time" backup, either to disk or to tape, or both. An important point is that copy and archive policies only duplicate data to another container; they do not remove or modify in any way the original file. They do however add new attributes to the file. Those attributes are pointers to the copies of the file, and to the older versions of the file.

2.1.3. Primary Event Daemon

File system integration is accomplished with the primary event daemon, hsmd. It can currently be compiled to support either the SGI XFS DMAPI or the IBM JFS DMAPI library. We anticipate supporting other DMAPI enabled file systems in the future.

The hsmd detects all events related to creation or modification of managed files, and notifies any registered policy engines via UNIX domain sockets. One hsmd instance handles the events for a single managed file system. Multiple managed file systems would require one hsmd instance for each.

Write events get special handling because DMAPI currently has no "close" event, and because for large files there can be thousands or millions of write events (which would pose performance and resource problems if each of these events had to be handled by the user level hsmd code).

Thus, upon receipt of a write event, hsmd places the event descriptor on a "changelist", and turns off write events for the subject file. The changelist is checked at tunable intervals, and any files that have not changed within the tunable change interval are removed from the list and any registered policy engines are notified of the change to the file. Write events are re-enabled on the file before notification of policy engines, guaranteeing that if the file subsequently changes, another event will be generated. This allows policy engines to either take action or drop an event if a file has changed since the timestamp of the event, knowing that another notification will be forthcoming when the file ceases changing.

The hsmd also supports the standard HSM activity of punching holes in files (removing a file's data while maintaining application-transparent referential integrity). If a read event occurs on a file whose data has been removed, hsmd prevents the I/O from proceeding while it forwards the event to the policy engine for data retrieval. The policy engine will respond to the event when the data has been restored, allowing the I/O to resume in an application-transparent fashion. The policy engine can also cause the application to error out, if the data is unavailable or inaccessible according to policy.

2.1.4. Policy Engines Overview

The hsmd communicates with policy engines via UNIX domain sockets. Current code supports up to four simultaneous socket connections to policy engines per hsmd instance. This limitation is arbitrary, but sufficient for most or all environments (the code can be recompiled to support more policy engine connections).

We have implemented two primary policy engines, and the architecture supports creation of other policy engines as needed. Each policy engine is responsible for the following:

- Applying policy as appropriate to its function (normally this would mean copying new or changed files to some alternate container, but might be almost anything).
- Attaching attributes to files as necessary so that copies made *by this policy engine* can be located and retrieved or updated.
- Servicing data faults, if permitted by the policy.
- Update the central SQL metadata database, if in use.

Associated policy engine utilities must also:

- Report the location of any copies of the file *saved by this policy engine.*
- Verify and report whether there is a non-stale copy of a given file (for use in permitting hole punching).
- Audit a file system to determine whether there are any files for which policy has not been applied.

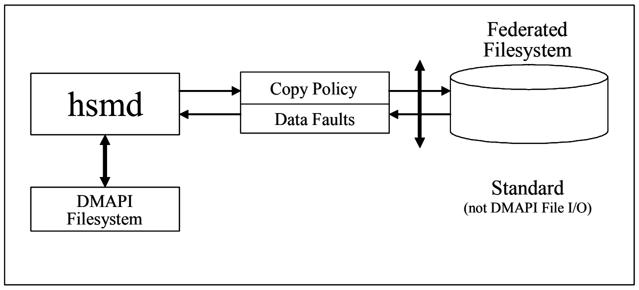

Figure 2. Copy Policy Data Flow

Primary policy engines may support one or more secondary policy engines, as is the case with the Database Policy engine, documented below. The interface to the secondary policy engines is a design point for the primary policy engine that supports secondary engines.

2.1.5. Copy policy

The first policy engine is a copy policy, and its function is to duplicate files to a secondary “federated” file system as the files appear in the primary file system. Only the primary file system must be DMAPI enabled (unless policy will also be run on the secondary file system as well). Copies are written to the secondary file system using standard file I/O for writes, and for the reads necessary to service a data fault on the primary file system.

Since the tier 2 file system need not be DMAPI-enabled, any file system will do, including one mounted by NFS. The disadvantage is that we can’t maintain identical file attributes (specifically create, change and access times).

Future plans include creating a new copy policy engine that will work on a client/server basis when the tier 2 file system is DMAPI-enabled, allowing complete consistency of attributes between tiers. In the mean time, standard file I/O works well aside from the minor points noted.

Our approach has some compelling advantages over block level device mirroring, as the target file system is independent of the primary from a validity and consistency standpoint. This means of duplication does not propagate file system corruption, nor does it depend on the in-order completion of I/O for a valid secondary file system. Asynchronous mirroring at a file level avoids the data integrity issues associated with block level mirroring commonly applied in storage subsystem hardware.

A copy policy implementation with rich data classification capabilities (e.g., more than one duplication target selected based on file type, ownership or other attributes) is planned for the future, as a secondary policy engine to the Database Policy engine.

2.1.6. Database policy

The Database Policy engine (db_policy) is designed to reliably support more complex and sophisticated policy engines by providing the event stream as a SQL based event queue. Events received from hsmd are stored in a persistent RDBMS based work queue (the Primary Work Queue), and a secondary event handler is used to drain the queue and act upon the events. This approach avoids both the memory management limitations and the lack of persistence across boots that affect an in-memory event queuing approach.

Advantages of this approach include the following:

- Work queues are persistent.
- Off the shelf cluster database servers can be utilized to easily architect multi-node, multi-tier implementations that deliver high availability characteristics.
- New secondary policy engines may be quickly prototyped in any scripted language that has good database support, such as Perl. This has been important in designing data classification practices where data management logic can quickly be tested for a good fit with the "business logic" of the problem to be solved.
- Work queues may be queried by any number of agents other than archive policy to achieve other objectives (for example, resource monitoring, QOS or fairness enforcement, or secondary work queue processors for high availability).

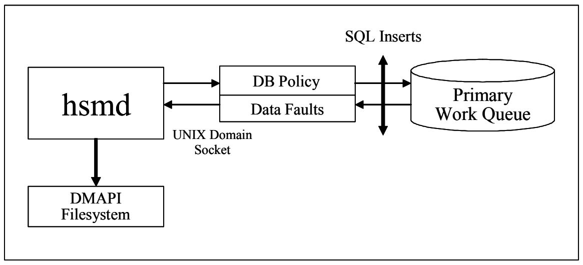

Figure 3. Database Policy Data Flow

Disadvantages of using the RDBMS might include scalability and footprint issues, but we feel these can be addressed with the distributed nature of modern RDBMS servers. Instead of running a database instance on each system with a managed file system, the database schema employed allows us to use one or many RDBMS severs for one or many managed file system servers. Maintaining database server integrity is a well documented subject, and the MySQL code we currently employ has a good reputation for stability.

2.1.7. Secondary Policy Engine: Archive Policy

The Archive Policy secondary policy engine removes events from the primary work queue database, and sorts them to an arbitrary number of policy queues which are serviced by their own data movers. This allows files meeting different criteria to be collocated on tape volumesets, separate from the data not meeting those specific criteria. This construct allows us to disaggregate data based on any desired criteria, and bring it into TMS as cataloged files under policy based management provided by the TMS.

As currently implemented, the Archive Policy engine uses regular expression analysis and file attributes to sort relevant events into work queues for separate data movers. Each data mover services a collocated set of files on tape. An example archive policy would act on files with a ".doc" extension. Those files would get archived to a volumset named "Word_Docs", with the option of having identically named volumsets for each user ID to separate files by both type and owner within the archive. The volumset's attributes might include an onsite or offsite location, tape technology type, and an attribute to keep the last five generations of files.

We know the difference between a file create and a file modify, by virtue of storing the TMS reference for the file within the file, as a user attribute when it is archived. When a file is modified, archive policy will append another file to the volumeset with the same fileid as the previous generation, and an incremented generation number.

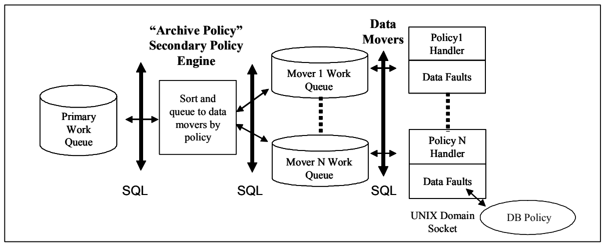

Figure 4. Primary and Secondary Policy Engines

The various generations of a file continue to add new TMS references within the file's user attribute area, supporting rollback of the file system state to restore individual files back to previous generations.

2.1.8. Metadata Database and Block Release

The Archive Policy engine also maintains information on its actions in the metadata database, which contains key attributes of files. This database is useful in locating all copies of a file, or in finding candidates for hole punching or permanent deletion, or just general analysis of data and filespaces (without dragging the whole file system structure through memory). OpenSMS does not rely on this database always being completely in sync with the file system state.

Like copy policy, the Database Policy / Archive Policy combination provides services for block release ("punching holes" in files), if appropriate. The metadata database can be queried by the block release tools to identify block release candidates. For example, a select sorted by size and last access or perhaps a user defined criteria such as a low "quality of service" rating.

Once the list of files that satisfy the query is complete, the candidate files are passed to a utility that verifies that we have a non-stale copy, either in a downstream federated file system or the TMS archive, and releases the data blocks if so. Determination of the existence of the non-stale secondary copy is based on the copy pointer in the file's user attributes having a timestamp that is later than the ctime for the file object.

One important aspect of the metadata database is that it is not an essential component in our data management system. While its data integrity is assumed not to be problematic, if it is somehow compromised and needs to be resynchronized, only the RDBMS selection of block release candidates will be affected during this process. Referential integrity needed to service a data fault is still good, since we store the TMS catalog entry for the file as a user attribute within the actual file system, as well as in the file system dump file.

2.1.9. OpenTMS

OpenTMS brings standard tape I/O subsystem services and file cataloging. It is code that has historically been ported to virtually every operating system. It performs functions such as device allocation, mount request processing, low level device handling, media management, a vault management system, and services for direct tape I/O typically only found in mainframe operating systems. By providing high-level services for direct tape I/O, OpenTMS allows enterprise class hardware to be shared by disparate applications and hosts, making highly reliable hardware cost effective by enabling high device utilization rates.

While OpenSMS manages the policy-based movement of data files between a DMAPI compliant file system and OpenTMS, once files have been copied to OpenTMS, they become directly addressable objects. This is a common data access method in enterprise computing on large systems, and allows direct access to tape, bypassing disk altogether. We use these combined file system/TMS access methods often in a hybrid manner, for example dropping a video file into a network exported file system, and then later accessing it for viewing or transcoding directly from a TMS request for the file.

As previously mentioned, OpenTMS volumesets are created with specific policy attributes that are used by the TMS to manage their life cycle. Included in these policy attributes are the expiration attributes. Expiration of data once it is copied into a volumeset can be based on criteria that include:

- Time since creation
- Time since last access
- Version number (keep N versions before expiring the oldest)
- Generation number (keep N generations before expiring the oldest)
- Never expire unless explicitly done by operator action

If a file object exists only as a TMS cataloged object and a user or application attempts to open the object, it is copied back into the file system where the open occurred (via DMAPI), and then the application I/O is satisfied normally. If the TMS object being called has been expired, but not yet overwritten, the file system open will still be satisfied.

The interesting concept that the TMS brings to the table is the association of policy attributes with the volumesets. The attributes control the technology used for storing the data (and the associated performance and reliability), the physical location for storage, whether there is a scheduled movement for the container, ownership and disposal attributes, and of course retention attributes.

When we combine the concept of policy attributes of TMS storage, with the highly granular classification of data that is available with database policy engines, we are able to create a complex and self-managing multi-tier storage system. This multi-tier architecture, driven by the ability to disaggregate data on very minor distinctions, has implications; particularly in terms of the removable media handling, and these need to be taken into account.

An example is creation of policies that aggregate data by user or group id, after disaggregating by file type. This type of a deployment is attractive in terms of the way one might like to view the collocation of user data into separate removable volumes. It also will drive an amazingly high library mount rate as different users create files with different applications, or worse yet does a recursive copy of a large directory tree off of a multi-user server into a managed file system.

2.1.10. TMS Library and Tape Drive Interfaces

Library interconnectivity is affected through site exits, which exist for each tape drive in the form of mount and unmount executables for that drive's associated media changer or library. These executables can be run on any named TMS netclient. If the netclient isn't named in the site exit file, the executable will be invoked on the netclient where the device allocation is made. This allows for the library control to be distributed in an environment where it is implemented in a shared SAN, or an IP based library interface. We use both environments. The IP based implementation uses StorageTek's UNIX based library control platform, ACSLS. In a direct SCSI library environment, we use site exits based on the mtx SCSI media changer project. A minor change had to be made for mtx to work, the SCSI command descriptor block for the unload command had to be changed to force the rewind-unload operation at the tape drive through the media changer interface. The reason is that the TMS close of a volumset leaves the drive loaded and in a ready state in case another subsequent user access could be satisfied with the existing mount.

Historically the tape drive interface has always been challenging because of the many different device attributes presented by the different tape technologies on the market, and the fact that the TMS needs to drive all these devices capably and with the best performance possible. In order to accomplish this, a tapecap file has been employed, and it serves much the same purpose as the /etc/termcap file serves for terminal I/O.

Within the tapecap file, the entire device attributes and appropriate IOCTL's are listed. This file will have drive entries for each device, and its driver/OS environment so that drive specific attributes such as inter-record gap can be accommodated alongside driver/OS specific attributes such as whether or not end of tape is supported, or must be calculated. Populating this set of attributes is done via a utility which exercises the target tape drive, and discovers its tapecap profile.

2.1.11. TMS Namespace Considerations

The process of conveying file objects from a hierarchical file system namespace to a flat TMS namespace imposes some interesting challenges. Having many files from different file systems and directories with identical names in a single TMS volumset is possible, file ids do not have to be unique within a volumset. However, it imposes the requirement that the file sequence number be used as the unique file descriptor. The problem here is in that if we were to use the file sequence number, we would then have to update the file system pointers to do either reclamation, or media transcription.

That would be very undesirable, so we create a unique file id, and preserve the original file name (at least the first 40 characters) within the TMS catalog as the file comment. This produces a very readable volumeset listing, since the file comment appears with each file entry. What is seen in a volumset listing as files are created and modified over their life, is a contrived file id that is unique to a specific name and path in a file system. That file id will be repeated for every modification of the file that is archived, with each having an incremented file generation number.

The volumset will typically have a retention policy that keeps N generations of a file. As soon as enough generations of a file have been archived, the older ones will begin to be expired. Reclamation simply calculates the percentage of expired vs. unexpired blocks on a volumset to determine if the reclamation threshold has been met. If it has, reclamation will create a new volumset with the same name as the source volumset, but an incremented generation number.

So we use the generational construct in two different contexts. In a file context it is used to rollback to a previously archived file object generation. In a volumset context, it is used to copy a fragmented volumset to a new unfragmented volumset, or to transcribe an old volumset to a new media or another type of removable technology.

Since we maintain the uniqueness of the contrived file id within the volumset, we can do the volumset transcription or reclamation without changing any file system attributes, and still have data fault servicing work as it should. We haven't saved any pointer to the archive copy that has a context related to the physical media, as we would have if we had used a block id or file sequence number.

Another consideration is the application of user attributes, and their preservation as the file is archived into the TMS namespace. We chose not to create any user attributes in our environment that have a scope outside of the file system because of the difficulty of attaching them to a file as it leaves the file system. Standard file attributes match well with the metadata that exists in a standard mainframe compatible HDR3 tape label, so we write one of these with every file. This makes a volumset completely portable, since a TMS client can scan an uncataloged volumeset, and recover all the file metadata necessary to repopulate the TMS catalog.

Other user attributes that might be helpful in the application environment would be problematic. The file object could be encapapsulated within a dump file, and the user attributes would be preserved. However portability would be limited, since a user recalling the object from the TMS on another platform may well not have the appropriate restore utility.

As it stands now, a file object appearing in a managed file system gets archived in a platform independent, standard labeled tape. A TMS client on any platform can access that file object directly, with no post processing required to convert it back to the original file object. We view this preservation of a platform independent archive, with all metadata preserved in standard labeled media as unique, and quite useful.

2.1.12. TMS Interfaces

The TMS resident file objects can be accessed at a command line interface (or within a shell script), or through a C API. Accessing files through the command line is done through a set of commands that allow the user to view the file contents of a volumeset, and then request the object with a read on the selected file. Command line options exist for use of standard I/O redirection or named pipes to avoid bringing the object back into the file system in applications where the file is being brought back to be processed and then saved in some transformed state, such as parsing a large body of data for insertion into a RDBMS system. Creation of the volumset with its policies (retention, technology, location, scratch pool, etc…) is the first step in saving TMS file objects. Once a volumset is created, it must be accessed in write mode, and then files are saved using either recputs in the API, or file I/O, named pipes, or I/O redirection at the command line.

2.1.13. TMS Scalability

As previously stated, we have used a legacy enterprise product with an extensive history as our TMS. This gives us high confidence in the stability of the code base within its design limitations, which it 6 million file objects, 4 million volumesets, and 255 tape drives.

In order to get beyond these limitations, we are looking at two changes to the existing code. The first is to replace the existing B+Tree catalog code with Berkley Database code. We are also looking into modifying the TMS client to understand a volumset name that is qualified with a TMS catalog server name. Currently, each TMS client has a single master server defined in a configuration file. This latter change would allow us to work in an environment where we have an arbitrary number of TMS catalog servers, each with its own volumsets.

2.1.14. Tape Subsystem Considerations

This architecture imposes performance and reliability demands on tape automation that are significantly greater than in a traditional backup/restore environment. In the current implementation, files are individually cataloged in the TMS. This is done so that once there, the file objects have enterprise wide scope, and can be easily accessed directly from the TMS services on any client system, by any privileged user, without any post processing. We write the volumesets as standard ANSI labeled tapes, with a HDR3 label for each file.

One of the implications of this design decision is that we add about a second of overhead per file written for label processing (when using enterprise class tape drives). Writing tape labels involves writing tape marks, which forces tape drives to do some physical things that take time. Some backup oriented tape drives will handle this very badly, taking much more than a second or so per file.

We did this because we choose to use rich data classification to manage and archive only the files that were valuable to us, and ignore everything else. Using standard HDR3 label processing on a per file basis not only adheres to a decades old enterprise standard for data interchange on removable tape media, it allows us to bring a previously written volume into a TMS, and populate the TMS catalog by telling it to “scan” the volume, recovering all the file information from those HD3 labels.

Alternative approaches could be taken at the sacrifice of some benefits, such as logical collocation of data by file type. Archive policy can be very simple, putting all different file types together as they are created and changed onto a single volumset. This in itself will reduce the number of mounts and would be much less taxing on backup oriented hardware.

Increasing the archive rate for small files could be done by aggregating many small files into a few larger tar or dump files to address the label-processing problem, and it has been considered as well.

We chose not to go down either of these paths, since enterprise hardware that meets the demands of an active archive is readily available, and we viewed preserving these characteristics as highly desirable. The deciding factor in preserving label processing was the availability of a fairly simple hardware solution to the performance problem in a small file environment. Tape virtualization systems that write first to a disk buffer, then write large files to real tape drives should avoid that problem altogether.

2.1.15. Shared Active Archive

This type of “direct access to tape data” architecture requires reliable, enterprise class technologies to support the high mount rates and duty cycles. These drives and libraries are costly, and need to be shared resources to be cost effective as well as to facilitate data sharing at the TMS service level. OpenTMS is architected for dynamic drive sharing between dissimilar hosts and applications to provide these benifits.

The TMS catalog and device allocation functions reside on a master server, communicating with net clients that run on any node either using TMS services, and/or hosting a tape transport. OpenTMS makes these drives available to any network attached host, by means of either a network socket, or direct SCSI, or attachment through switched fibre channel fabric. For SCSI attached environments, transports would typically be distributed with larger numbers of drives on the hosts doing the most I/O, to keep the I/O off the network to the greatest extent possible.

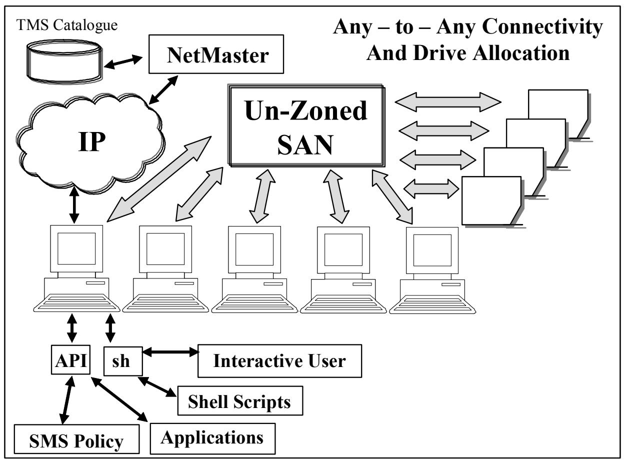

Figure 5. Shared Archive Architecture

When using enterprise class drives, typically they would have fibre channel interfaces and be attached to an un-zoned fabric, creating a device entry for each transport on every fibre channel attached host. Taking advantage of this any-to-any connectivity is supported by configuring the master server with an arbitrary name for each transport. Each transport name then has an entry for each fibre attached host, including its */dev/rmt* entry. Allocation between hosts is dynamic, and will be by default preferenced by allocation of a channel rather than network attached resource. The TMS also accepts a "*machine=*" attribute if the user chooses to have control over data path allocation for some reason.

Accessing cataloged objects in OpenTMS can be done via user scripts with standard I/O redirection, named pipes, or through a C language API.

2.1.16. Data Protection Issues

Protecting files with archive policy doesn't in itself alleviate the need to protect the file system space, so we still have to contend with dump/restore. Fortunately, the problem of large dump files can be cleverly avoided.

SGI's xfsdump utility has a command option to ignore the user data in a file if it has an attribute that indicates it is "dual state," meaning it has previously been archived. These files will still be present in the dump file, but only in the form of the standard and user metadata, the later containing all the information necessary for data fault handling from an archive copy.

We could envision some fairly capable data protection schemes based on the available tools with a managed file system. One example might be to run a utility on a system as soon as the operating system installation is completed off of the distribution media to set every file as "dual state". One would then set up archive policies to make archive copies of every file created that is of some long term value, and ignore everything else.

This would provide near real time backup of user data, and would allow routine dumps of the file system containing only file system metadata. Backing up operating system files that are installed during a bare metal recovery would be avoided. Restore could be done very quickly by first restoring only the file system metadata, and then allowing open events to trigger repopulation of user data into the file system.

3. Related work

We looked at several other Hierarchical Storage Managers over the years and learned a few lessons that we tried to incorporate into OpenSMS. But we also looked at Systems Managed Storage as a more comprehensive issue in terms of the added requirements of integrating regulatory compliance, offsite archiving, and asset management (specifically, easing the retirement of large disk arrays).

One of the differentiators of OpenSMS is the concept of bringing a TMS into the environment as an alternative to disk storage altogether. There are many data access requirements that don't match the attributes of disk I/O well, and shouldn't be satisfied with disk storage. Batch processes that do large sequential I/O are good examples. These processes put a significant load on a file system, denying performance to applications that need low latency, or bursty I/O. Arguably, SGI's DMF when combined with TMF and OpenVault have many of the features of OpenTMS, but also a few shortcomings. Specifically, there does not appear to be any provision for direct access to data from tape if it came to the tape system through the HSM daemon[5], and referential integrity is dependent on the consistency of a meta data database.

SamFS from Sun has very good scalability because they store all the pointer information needed to maintain referential integrity back to the offline storage within the inode. This makes an inode dump and the offline copies the only information needed to restore a file system, and that is a key feature we decided we had to have. DMF and ADIC's StorNext Management Suite both use relational databases that have to be kept in sync with the inodes, and offline media, and we view this as problematic in that the referential integrity of file system pointers and offline copies depends on too many things being all right, all the time.

On the down side, SamFS uses tape hardware that is captive to a single server, which is a common problem with most applications on client server platforms. Having to dedicate tape drives on a per server bases, with no application sharing makes cost justification of reliable enterprise hardware very difficult. Using low cost transports and libraries for active HSM is an unreliable design choice in our experience.

SamFS also doesn't have a user accessible TMS. It writes tar files, but direct access to tape data isn't really supported in anything other than a manual process. SamFS also has primitive support for an inline disk tier. It doesn't store files in the secondary disk tier as standard files, they are stored as tar files. OpenSMS allows the secondary disk tiers to work as normal disk mirrors to facilitate unmounting a primary, and remounting a mirror in its place. This was done principally to facilitate retirement of obsolete disk technology without the down time associated with data movement. We haven't seen this capability explicitly noted in other HSMs. It is unclear to us whether or not DMF has this capability.

4. Future work

We expect to create some new variations on copy policy to add the same rich data classification that exists currently in archive policy. Additionally, a copy policy that works in a true client-server fashion with another DMAPI enable file system is being considered.

We also anticipate other policy engines, some that might make use of an ftp repository for example. Some other possibilities outside of storage management may also make sense. We could for example use the appearance of a certain file type in a managed file system to execute some batch process, such as a transcoding an mpeg2 file into an mpeg4, or automatically indexing files as they are created.

Data protection that is completely integrated into OpenSMS is a high priority. Since it needs to be integrated into whatever dump/restore utilities are provided by the file system vendor, we'll need to come up with a flexible approach to solving this problem.

5. Conclusions

So far, the free market has failed to solve the data management problem in an effective manner. The existence of many proprietary variants of UNIX, all similar, and yet all different, has made the task too challenging for anyone hoping to provide data management where it is needed - at the file system level.

The emergence of Linux as an open-source alternative provides another chance to address this problem in a comprehensive manner. We believe OpenSMS can bring systemic management of data, as practiced for many years in mainframe computing, to the client-server platforms of today's enterprise environment.

OpenSMS will surely need time to evolve into a data management system with enterprise class reliability, but we think an open source model will serve this need well. We also feel the lack of licensing costs and access to sources will give OpenSMS a leg up in establishing a platform to develop some badly needed standards for storage management.

References

[1] Michael Peterson, Information Lifecycle Management - A VISION FOR THE FUTURE. *(http://www.sresearch.com/strat_profiles/SRC-Profile%20ILM%20Vision%203-18-04.pdf)*

[2] Peter Lawthers. The Data Management Applications Programming Interface. (Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems, pages 327-335)

[3] Jim Mostek, William Earl, Dan Koren Russell Cattelan, Kenneth Preslan, and Matthew O'Keefe. *(Porting the SGI XFS File System to Linux. http://oss.sgi.com/projects/xfs/publications.html)*

[4] Steve Best, Journaling File Systems - Advanced Linux file systems are bigger, faster, and more reliable *(Linux Magazine, October 2002)*

[5] Neil Bannister and Jim Mostek, DMF Overview, September 1999 *(http://oss.sgi.com/projects/xfs/publications.html)*