[illegible] /S 117700

N95- 24120

Integrating UniTree with the Data Migration API

David G. Schrodel
Convex Computer Corporation
3000 Waterview Parkway
PO Box 833851
Richardson Texas 75083-3851
Tel: +1-214-497-4565
Fax: +1-214-497-4500
schrodel@convex.com

S/2-61
43456
p- 19

1.0 Abstract

The Data Migration Application Programming Interface (DMAPI) has the potential to allow developers of open systems Hierarchical Storage Management (HSM) products to virtualize native file systems without the requirement to make changes to the underlying operating system. This paper describes advantages of virtualizing native file systems in heirarchical storage management systems, the DMAPI at a high level, what the goals are for the interface, and the integration of the Convex UniTree+ HSM with the DMAPI along with some of the benefits derived in the resulting product.

2.0 Introduction

For years developers of Hierarchical Storage Management (HSM) systems have had to choose between integrating their system with the underlying Operating System (OS), or take the more "open systems" route of not requiring any changes to the OS. Integration with the OS allows an HSM to virtualize the native file system. The open systems approach allows easier porting from platform to platform.

The advantage to virtualizing the native file system is that most all applications that reside on top of the file system will continue to operate without requiring changes. Even low level functions like the NFS[1] network protocols continue to function on HSM controlled file systems. In addition, applications that access data in these file systems can expect performance equivalent to file systems not controlled by an HSM for resident files. Examples of products in the market place today that require changes to the OS include the

[1] NFS is a registered trademark of Sun Microsystems, Inc.

EpochServ[2] product from Epoch[3] Systems Inc., and the AMASS[4] product from Advanced Archival Products Inc.

The Epoch Serve product uses "hooks" in the OS via a special device driver and changes to the native file system to gain access to events like file creates, removes, accesses, etc. Although the OS changes have been minimized, subtle changes in the operating system from release to release can introduce problems in the HSM support functions. For example, suppose the OS vendor changes the interface to the kernel memory allocator that the HSM hooks use. Now the HSM must be modified to work with the new allocator, qualified, and released with all of the costs associated with that process.

The AMASS product installs a new file system into the host operating system at the Virtual File System (VFS) layer. Although the VFS interface is fairly well defined industry wide, subtle changes from release to release can cause problems interfacing with the kernel support functions not as well defined. Although changes to the OS can be minimized, kernel integration makes porting of the products more difficult.

UniTree[5] is an example of a product that requires no operating system changes. It sits above the OS and accesses the required OS functions via the POSIX p1003.1 interfaces. It provides access to the data in the archive via the FTP, RCP, and NFS protocols. It does not use native file systems or utilities, but instead provides separate implementations of each. As a result, the product is more easily ported from platform to platform and usually does not require changes as a result of a new OS release. This portability comes at a cost however. UniTree products do not benefit from enhancements provide by the base operating system. New operating system releases tend to have both functionality and stability enhancements included. Because UniTree implements its own file system, the new functionality may not be available. Applications that access a file system by read and write system calls cannot access data stored in the UniTree file system without going through the NFS protocol stack. The access methods provided by UniTree tend to be slower than the same services provided by the underlying operating system. To quantify the performance penalty for this, a single stream access to the local file system in the ConvexOS can reach rates above 45MB/s. UniTree file system access through local host NFS protocols are under 1MB/s.

An alternate approach to the two type of HSM's described above is one that requires no operating system changes yet has access to the native file systems through a set of kernel

---

[2] EpochServ is a registered trademark of Epoch Systems Inc.

[3]Epoch is a registered trademark of Epoch Systems Inc.

[4] AMASS is a trademark of Advanced Archival Products Inc.

[5] UniTree is a trademark of UniTree Software Inc.

supplied "file system hooks". Three examples of HSM's with this attribute is the FileServ Software[6] product from EMASS[7], the Convex Storage Manager (CSM) product from Convex Computer Corp., and NAStore developed by NASA Ames Research Center. All of these products sit on top of the ConvexOS operating system which runs on the C-Series architecture's. ConvexOS exports a set of interfaces that provides functionality similar to what the kernel intrusive HSM's described above export to their user level applications. An application can receive events like read(), write(), trunc(), create(), etc. as well as suspend access to data in a file for non HSM applications. The HSM applications can read data from a file without updating time stamps, punch holes (i.e. free space in a file without changing the apparent size of the file), fill files with data previously migrated out, and re-enable access to the file for non HSM applications. All of these operations can be accomplished completely transparent to normal applications. This allows development of HSM applications like the kernel intrusive ones described above without the drawbacks of having to integrate changes into the base operating system.

The major drawback to the ConvexOS file system hooks are that they are not available on any platform besides the C-Series line of products. What HSM vendors require is access to a standard set of file system hooks across a diverse set of operating systems and platforms. Without this, an HSM vendor can only make a business case to support platforms where they can deliver sufficient volume of product to offset the inherent cost of supporting kernel modifications. This realization was what prompted a set of competing vendors of computer platforms and HSM products, to form an industry consortium known as the Data Migration Interface Group (DMIG).

3.0 DMIG

The DMIG consists of a large group of vendors and a smaller group of active participants in the specification process including: 3M, ACSC, Amdahl, Auspex, Avail Systems, Bull, Convex, E-Mass, Epoch, Hitachi Computer, HP, IBM, Lachman / Legent, Legato, NASA / Ames, Netsor, Novell / USL, OpenVision, SCO, SGI, Sunsoft, and Veritas[8]. This group got together through the 1993 - 1994 time period to produce a specification for a file system interfaces that all parties could agree to support (if not in a product, at least in spirit). The goal of the interface is to enable development of HSM and backup products on computer systems that virtulize the native filesystem without requiring OS modifications. Needless to say, there were many heated debates during the course of the meetings and on the DMIG reflector, but the group did finally agree on a set of interfaces known as the DMAPI.

[6] FileServ Software is a registered trademark of E-Systems.

[7] EMASS is a registered trademark of E-Systems.

[8] The companies listed are those that attended at least one meeting in the 1994 time frame and if I missed someone, I apologize in advance.

4.0 DMAPI [1]

The DMAPI is a set of interfaces to be provided by the base operating system that enables HSM and backup applications to gain access to native file system data and metadata transparent to normal applications. It includes the following basic concepts:

1: *Events* - Data Management (DM) applications can request to be informed of specific events like read(), write(), etc. The event notification is via messages which DM applications gain access to via the dm_get_events() call. There are two distinct message types; synchronous, and asynchronous. Synchronous messages have tokens associated with them. Asynchronous do not have tokens associated with them.

2: *Tokens*- a token is a reference to the state associated with a synchronous event message. The contents of a token are opaque to the DM application. DM applications can modify the contents of a token only through the dm_request_right() and dm_release_right() call. Rights can be granted to a DM application including DM_RIGHT_EXCL, DM_RIGHT_SHARED, and DM_RIGHT_NULL. DM_RIGHT_EXCL prohibits any access to a file except through DMAPI calls that accept tokens, and only then if the token with the DM_RIGHT_EXCL right is passed to the interface. DM_RIGHT_SHARED protects against any modification of the data or metadata associated with a file, by normal or DM applications, but will allow multiple accesses to the data or metadata. DM_RIGHT_NULL grants no rights.

3: *Managed Regions* - A managed region designates the portions of a file that the DM application is managing. There are 3 possible events that may be enabled on a managed region; DM_REGION_READ, DM_REGION_WRITE, and DM_REGION_TRUNCATE. If the corresponding action, read, write, or truncate, is attempted by a non DM application to the associated region, an event will be generated for the DM application that has expressed interest in the corresponding portions of the file. The number of managed regions supported by the DMAPI is implementation defined.

4: *Handles* - Pathname independent references to file system objects. A file handle uniquely identifies a file system object. Handles exist for file systems, directories, files, symlinks, and one special global handle that does not refer to any object but allows a DM applications to register for mount events.

5: *Sessions* - The interface between the DM application and the DMAPI is session based. The session is the identifier used to determine who receives events for a particular file system object. Sessions can be thought of as two things; a queue that event messages are queued ut to, and an identifier for use in tracking, auditing, and controlling access to the DMAPI facilities.

6: *Data Management Attributes* - Space for a DM application to store pertinent information about a file. The data in the attribute space is opaque to the DMAPI implementation. Examples of information that might be stored include the location of data from the file in the archive for migrated files. DM attributes are an optional portion of the DMAPI.

7: *Holes* - Holes can be created two ways. First is via the lseek() function. If an application seeks past the existing end of a file, and writes some data, implementation may create virtual space in a file without any corresponding storage allocated. The second way is via the DMAPI call dm_punch_hole(). This call frees the storage associated with the portion of the file where the hole is punched (i.e. frees file system space once a file is migrated to tertiary storage).

In general DM applications create a session, register for mount events, catch mount events and establishs interest in files by registering for events within a file system. They then catch events for the files of interest, and perform actions to migrate files in or out as needed.

DM applications are viewed as a part of the file system implementation and as such have no security restrictions placed on them. All calls to the DMAPI are expected to be run as superuser or some other file system permission that would give an application un-restricted access to the data and metadata in a file system. There is no association of DMAPI objects to any one process. Any application, running at the appropriate security level, with the correct session identifier and token identifier, can take actions on file system objects. The rational for this is to allow a DM application to use as many processes as required to deliver acceptable performance. There is also an underling assumption in the specification that only one DM application will attempt to control a file system. There is nothing to prevent multiple DM applications from controlling a file system, but coordination of the DM applications is outside the scope of the DMAPI. An example of a case where multiple DM applications may run on a single file system is where an HSM application and a Backup application cooperate to backup the HSM controlled file system. The reason for this is, for more than one application to control objects in a file system would require considerable machinery in the DMAPI. One of the primary goals of the DMIG group was to produce a specification that could be implemented in a six man month development project. To produce the machinery required would extend the level of effort far beyond the six man month goal.

The following are examples of DM applications pulled from the DMAPI specification. These are included to give a better idea of how the interfaces are intended to be used. [1]

4.1 Stageout

This example will stage out a 512 megabyte file names `/test/foo`. The first 64K will remain as a fence post. The remainder of the file will be staged out in 32 megabyte clusters.

The following concepts are illustrated:

• Use of the file change indicator

• Invisible read

• Setting managed region

• Punching holes

```
char            *buf;
void            *hanp;
size_t                hlen;
size_t                retrgns;
size_t                nchunks;
int                   change_end, change_start,
off_t           off;
dm_token_t      filetoken;
dm_stat_t       statbuf;
dm_region_t           rgnbuf[2];
dm_size_t       roff;
dm_off_t              rlen;
dm_sessid_t     sid;

if (dm_init_service() == -1) {
        err_msg("Can't initialize DMI\n");
        return(0);
}

dm_create_session(DM_NO_SESSION, &sid, "generic_app");
dm_path_to_handle("/test/foo", &hanp, &hlen);

/*
 * In order to stat the file, we need a shared lock.
 */
dm_create_userevent(sid, 0, (void *)0, &filetoken);
dm_request_right(sid,hanp,hlen,filetoken,DM_WAIT,DM_SHARED);

/*
 * While writing the file out to tertiary storage, we drop locks. We first
 * get the file change indicator, and query it after we're done.
 * If it changed, then we give up
 */
dm_get_fileattr(sid, hanp, hlen, filetoken, DM_AT_CFLAG, statbuf);
change_start = statbuf.dt_change;

/*
 * We don't bother with any DM attributes, just the data.
 * We write the file out in 32 meg chunks.
 */
nchunks = (512 * 1MEG) / CHUNKSIZE;
for (i=0; i<nchunks; i++) {
        dm_read_invis(sid, hanp, hlen, token, off, CHUNKSIZE, buf);
        dump_data_to_archive(hanp, hlen, off, buf);
        off += CHUNKSIZE;
}
dm_release_right(sid, hanp, hlen, filetoken);

/*
 * Store an DM application specific information, such as the file size,
 * file handle, etc., with the data
 */
dump_myinfo_to_archive(hanp, hlen);
```

```
/*
 * Check the file change indicator to see if the file changed
 * while we were doing other things. If not, then set a managed
 * region on the file
 */
dm_request_right(sid, hanp, hlen, filetoken, DM_WAIT, DM_EXCL);
dm_get_fileattr(sid, hanp, hlen, filetoken, DM_AT_CFLAG, statbuf);
change_end = statbuf.st_change;
if (change_start != change_end) {
        err_msg("File changed, bailing...\n");
        do_cleanup();
        return(1);
}

/*
 * Set up the managed regions so that the first 64K won't cause events
 * to be generated, but a foray into the rest of the file will generate
 * events
 */
rgnbuf[0].rg_off = 0;
rgnbuf[0].rg_size = FENCESIZE;
rgnbuf[0].rg_flags = DM_REGION_NOEVENT;
rgnbuf[1].rg_off = FENCESIZE;
rgnbuf[1].rg_size = (512 * 1MEG) - FENCESIZE;
rgnbuf[1].rg_flags = DM_REGION_READ | DM_REGION_WRITE |
DM_REGION_TRUNCATE;
dm_set_region(sid, hanp, hlen, filetoken, 2, rgnbuf, &retrgns);

/*
 * Punch a hole in the file. We assume that we know that what the
 * rounding constraints are so that we don't have to do a dm_probe_hole()
 */
dm_punch_hole(sid, hanp, hlen, token, rgnbuf[1].rg_off, rgnbuf[1].rg_size,
                &roff, &rlen);
/*
 * We're done. Release the token
 */

dm_respond_event(sid, filetoken, 0,(void *)0, DM_CONTINUE, 0);
```

4.2 Stagein

This example will stage in the file that was staged out in the above example.

There is a master daemon that receives messages from the kernel, and sends them on to worker bees for processing. The master daemon is only monitoring the `read`, `write`, and `truncate` managed region events in this example for the filesystem `/test`. There is some magic here that is not shown. The master daemon knows about, and has access to, two other sessions that are used to perform event-specific handling. Information is shared between the master daemon and these other processes through some application-specific mechanism that is not shown. This could be shared memory, a socket, a well-known file, or any other mechanism. The details are left to the dazed reader.

The following concepts are illustrated:

- Sharing of tokens and sessions
- Setting event disposition
- A simple get_event loop
- Having a master daemon *move* an event to another session
- Complex lock upgrade
- Setting managed regions

```
extern dm_sessid_t  rw_sid, trunc_sid;

void                *fs_hanp;
void                *msgbuf;
size_t                      fs_hlen;
size_t                      msgsize, msgbuflen;
dm_sessid_t          sid;
dm_token_t           fs_token, newtoken;
dm_event_set_t       eventset;
dm_eventmsg_t       *msg;

if (dm_init_service() == -1) {
        err_msg("Can't initialize DMI\n");
        return(0);
}

dm_create_session(DM_NO_SESSION, &sid, "generic_app");

/*
 * Since we'll be communicating with other processes, we do some
 * magic setup to get their sids, establish communications channels,
 * etc. We could use dm_send_event(), stuff the data in a shared memory
 * region, open a socket, or whatever
 */
setup_communications(sid, &rw_sid, &trunc_sid);

dm_path_to_fshandle("/test", &fs_hanp, &fs_hlen);

/*
 * Now get a token and rights so that we can set the disposition
 * of events
 */
dm_create_userevent(sid, 0, (void *)0, &fs_token);
dm_request_right(sid, fs_hanp, fs_hlen, fs_token, DM_WAIT, DM_EXCL);

/*
 * Set the disposition of the events we want to monitor
 */
DMEV_ZERO(eventset);
DMEV_SET(DM_READ, eventset);
DMEV_SET(DM_WRITE, eventset);
DMEV_SET(DM_TRUNCATE, eventset);
dm_set_disp(sid, fs_hanp, fs_hlen, fs_token, &eventset, DM_MAX_MSG);

dm_release_right(sid, fs_hanp, fs_hlen, fs_token);
```

The master daemon now enters a simple loop where it will spend all its time. It simply asks the DMAPI for more messages and dispatches them to its worker processes.

```
/*
 * Find out the size of the largest message that can be delivered
 * on this filesystem. We use this to size an event buffer to get
 * an arbitrary (16 in this example) number of messages at the same
 * time.
 */
dm_get_config(fs_hanp, fs_hlen, DM_MAXMSG_SIZE, (long)&msgsize);
msgbuflen = msgsize * 16;
msgbuf = (void *)malloc(msgbuflen);

/*
 * Enter a simple loop, looking for messages. We don't worry about
 * resizing the buffer
 */
for (;;) {
        dm_get_events(sid, msgbuflen, msgbuf, &ret_msglen, 0, DM_WAIT);
        msg = (dm_eventmsg_t *)msgbuf
        while (msg != NULL) {

                /*
                 * For read and write events, we send them to other processes
                 * with 'well known' sids that are handling these things.
                 */
                if (msg->ev_type == DM_READ || msg->ev_type == DM_WRITE) {
                        dm_move_event(sid, msg->ev_token, rw_sid, &newtoken);

                } else if (msg->ev_type == DM_TRUNCATE) {
                        dm_move_event(sid, msg->ev_token, trunc_sid, &newtoken);

                } else {
                        err_msg("Unknown event type\n");
                        dm_respond_event(sid, msg->ev_token, 0,(void *)0,
                                DM_ABORT,EINVAL);
                        continue;
                }
                msg = DM_STEP_TO_NEXT(msg, dm_eventmsg_t *);
        }
}
```

The worker bee processes also do some initial setup, which won't be shown. For a simple stagein, we'll assume we've receive a *DM_READ* event on a managed region. We join our fearless process at the point in which it has received the event that was directed to it from the master daemon, and is starting to reload the data.

```
int                     change_start;
void                    *hanp;
size_t                  hlen;
size_t                  nchunks;
size_t                  retrgns;
dm_off_t                off;
dm_size_t               len;
dm_right_t              right;
dm_sessid_t             sid;
dm_stat_t               statbuf;
dm_eventmsg_t           *msg;
dm_data_event_t         *read_event;
```

```
dm_region_t                     rgnbuf[1];

msg = eventbuf;
read_event = DM_GET_VALUE(msg, data, dm_data_event_t *);

hanp = DM_GET_VALUE(read_event, handle, void *);
hlen = DM_GET_LEN(read_event, handle, size_t);

/*
 * Check to see what the rights are that came with the message. If
 * they aren't exclusive, we must go get them
 */
dm_query_right(sid, hanp, hlen, msg->ev_token, right);

if (right == DM_SHARED) {
        /*
         * We really need exclusive. We'll try to upgrade the lock,
         * but if that fails, we'll have to drop it and go to
         * sleep.
         */
        if (dm_request_right(sid, hanp, hlen, msg->ev_token, DM_EXCL,
                        DM_NOWAIT) == -1
        {
                if (errno != EAGAIN) {
                        err_msg("Can't upgrade lock\n");
                        do_cleanup();
                        return(1);
                }
                /*
                 * Before we drop the lock, get the file change indicator
                 */
                dm_get_fileattr(sid, hanp, hlen, msg->ev_token, DM_AT_CFLAG,
                        statbuf);
                change_start = statbuf.dt_change;

                dm_release_right(sid, hanp, hlen, msg->ev_token);
                dm_request_right(sid,fshanp,fshlen,msg-
>ev_token,DM_WAIT,DM_EXCL);

                /*
                 * Now that we've come back from sleeping, see if the file changed.
                 * If so, we just bail.
                 */
                dm_get_fileattr(sid, hanp,hlen,msg->ev_token,DM_AT_CFLAG,statbuf);
                if (statbuf.dt_change != change_start) {
                        err_msg("File changed. Bailing...\n");
                        do_cleanup();
                        return(1);
                }
        }
} else if (right == DM_NONE) {
        dm_request_right(sid,hanp,hlen,msg->ev_token,DM_WAIT,DM_EXCL);}
}
```

The worker bee is now at the point where it has exclusive access to the file. This is needed for dm_write_invis().

```
/*
 * Now that we have exclusive access to the file, we need to
 * go off and find where we stored the file's data in our
 * repository
 */
offset = DM_GET_VALUE(read_event, offset, dm_off_t);
len    = DM_GET_VALUE(read_event, len, dm_size_t);
find_our_file(hanp, hlen, offset, len);


/*
 * Restore the data for the file.
 * We'll assume that the file length is some nice integral multiple
 * of our chunksize;
 */
nchunks = len / CHUNKSIZE;
for (i=0; i<nchunks; i++) {
        get_data_from_archive(hanp, hlen, off, buf);
        dm_write_invis(sid, hanp, hlen, token, off, CHUNKSIZE, buf);
        off += CHUNKSIZE:
}


/*
 * Clear the managed region
 */
rgnbuf[0].rg_off = 0;
rgnbuf[0].rg_size = 0;
rgnbuf[0].rg_flags = DM_REGION_NOEVENT;
dm_set_region(sid, hanp, hlen, msg->ev_token, 1, rgnbuf, &retrgns);

/*
 * We're done. Release the token
 */

dm_respond_event(sid, msg->ev_token, 0,(void *)0, DM_CONTINUE, 0);
```

4.3 DMAPI Status

The current status of the DMIG is that the V2.0 version of the interface specification is out for industry review. Several companies are actively prototyping the interface, and some are close to having product available in the marketplace. Below is the status of several companies polled for their stance regarding the DMAPI:

1: Convex - Development under way for C-Series platforms for NAStore product. Development underway for HP platforms supported as data management platforms. Plans to port DMAPI to SPP product line in Q1-Q2 1995.

2: Hitachi Computer - Development underway to provide DMAPI on Veritas file system to integrate with Epoch product.

3: IBM - Prototyping underway. No firm plans to release support

4: SGI - Will release support for DMAPI in first half of 1995 in their XFS file system.

5: Sunsoft - Plan is to watch marketplace. No plans currently exist to support DMAPI.

6: Legent - "Legent, through its acquisition of Lachman Technology, has been behind the DMAPI interface ever since the group began meeting. We believe in the goals of the group and anxiously await adoption of the interface by UNIX operating system vendors. DMAPI will strengthen these platforms and will help vendors like Legent offer its storage management products across the broadest range of available systems."

Although not all vendors have committed resources to the DMAPI, it is clear from the above list, that several vendors believe that it is a viable interface and worth investing in. Given that the early adopters of the interface are successful in the market place, other companies are likely to put plans in place to formally support the DMAPI.

5.0 UniTree+ and the DMAPI

In section 2.0, several of the deficiencies in UniTree+ were described. It fundamentally is a result of the fact that UniTree+ does not virtualize the native file system. What was not discussed in section 2.0 was that UniTree+ has the capability to handle 1,000,000+ files and many terabytes of data in the archive. The combination of UniTree+ and an access method that virtualizes the native file system yields a product that provides the functionality and performance of native file system accesses, and the ability to support very large archives with high data ingestion and retrieval rates.

5.1 Standard UniTree+

UniTree+ as shipped today has an architecture that consists of a the following major blocks:

1: Nameserver - A server that maps pathname to capability ID. It supports a namespace that looks very similar to traditional UNIX[9] file system namespace.

[9]UNIX is a registered trademark of UNIX Systems Laboratories, Inc., a wholly owned subsidiary of Novell, Inc.

2: Disk Server - The server responsible for providing magnetic disk cache for data in the repository. All access methods read and write through the disk cache. No direct reading or writing of tape is supported.

3: Tape Server - The Server responsible for moving data to and from the tape system(s) and the magnetic disk cache. Provides mapping for capability ID to tape location mapping.

4: Access Servers - Daemons that provide the FTP and NFS protocols.

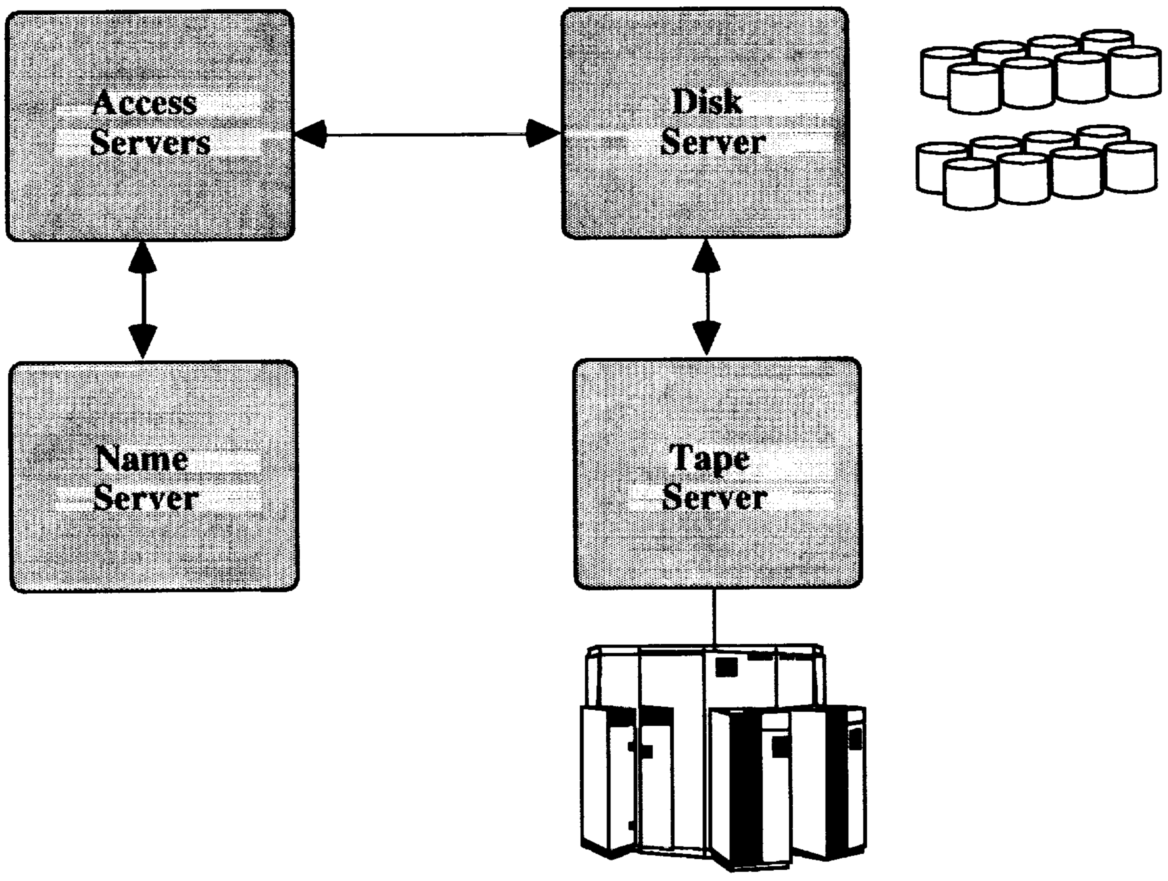

Figure 1: UniTree+ Today

The problem areas in the above architecture reside in the access servers and the name servers. All accesses traverse a network protocol. Even same machine accesses go through the localhost interface. The bulk of the slowdowns in the UniTree interface is because of latency associated with small transfers. This is especially true for metadata operations. If you view the graph below comparing native HP/UX[10] file system operations versus UniTree+ running on the same HP[11] system, you will notice a substantial difference in wall clock execution time. Times shown as .1 seconds returned zero seconds when measured with the HP/UX *time* command because the granularity of wall clock time is 1 second. There was no effort made to optimize the times shown below, but they are instead just an indication of relative execution times between native file system operations and UniTree+.

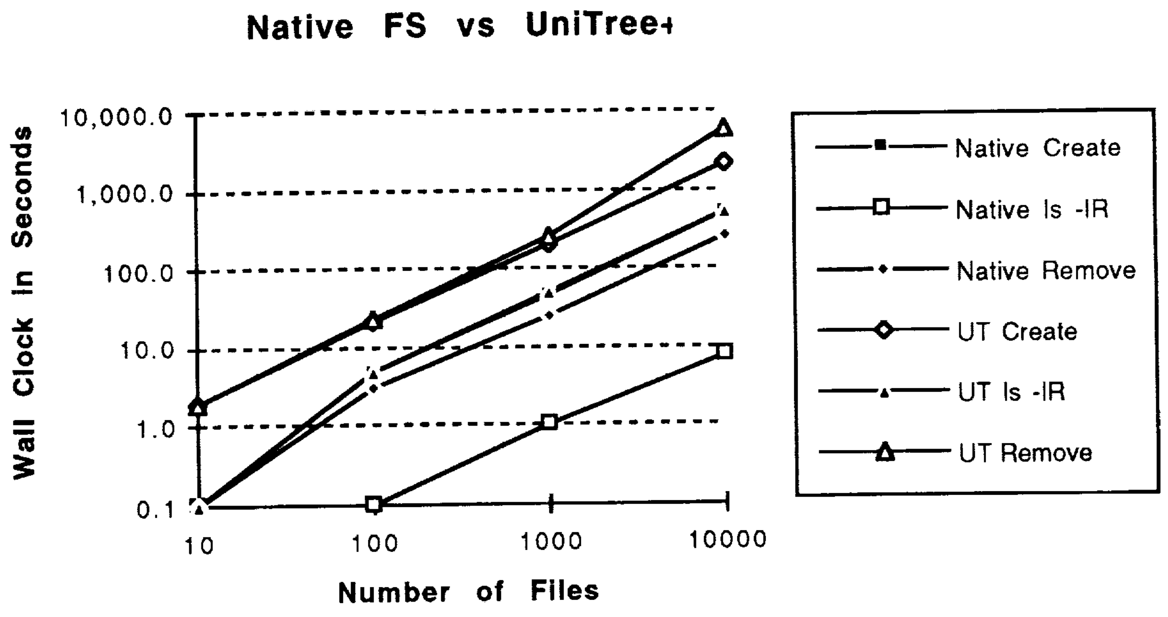

Figure 2: Performance Graph of Native File System versus UniTree+

All of the files created in the above test were of zero length. UT 10,100,1000,10000 refers to UniTree performing the operation listed on 10,100,1000,10000 files respectively, where Native 10,100,1000,10000 refers to the same operations executed on the Native file system and 10,100,1000,10000 files respectively. Creates of zero length files in UniTree+ involve primarily the NFS access daemon and UniTree+ name server. Replacement of those two servers in UniTree+ with native file system operations will dramatically improve the metadata operations in the resulting HSM.

[10]HP/UX is a registered trademark of Hewlett Packard Corp.

[11]HP is a registered trademark of Hewlett Packard Corp.

5.2 Convex Virtual Disk Manager (CVDM)

The Convex Virtual Disk Manager (CVDM) does exactly what the name implies. It's function is to control space allocation in native file systems using the DMAPI. It currently runs on ConvexOS and HP/UX systems controlling the UFS native file systems. The major functional blocks are as follows:

1: Migd - Responsible for startup and shutdown of system.

2: SSD - Responsible for creation of daemons for each file system put under CVDM control.

3: Migdmon - Daemon responsible for interfacing with the DMAPI to catch events and start actions required by the events.

4: Migout - Responsible for scheduling migrate out processing.

5: Migin - Responsible for migrating in data as a result of an access to non-resident data within a file.

6: Interface Manger - Responsible for communication with UniTree+ back end.

The structure of the daemons looks as follows:

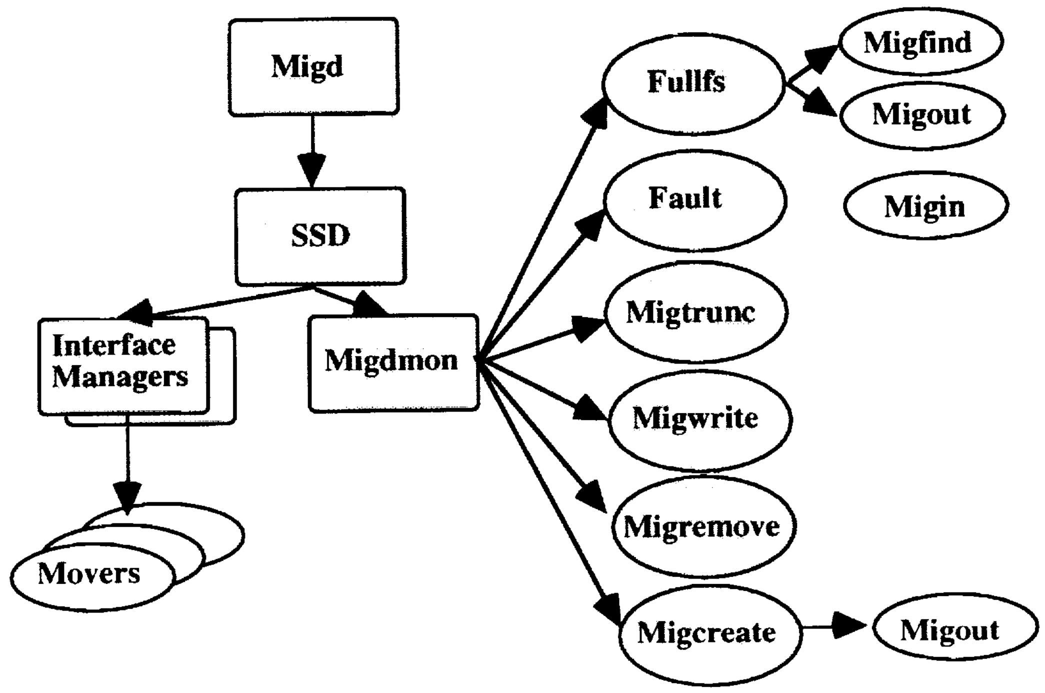

Figure 3: CVDM Server Architecture

The interface manager and movers are responsible for communicating with the UniTree+ back end. As a result of the separation of CVDM from the UniTree+ archive portion, multiple instances of CVDM can communicate with multiple instances of UniTree+. The following diagram illustrates a possible configuration:

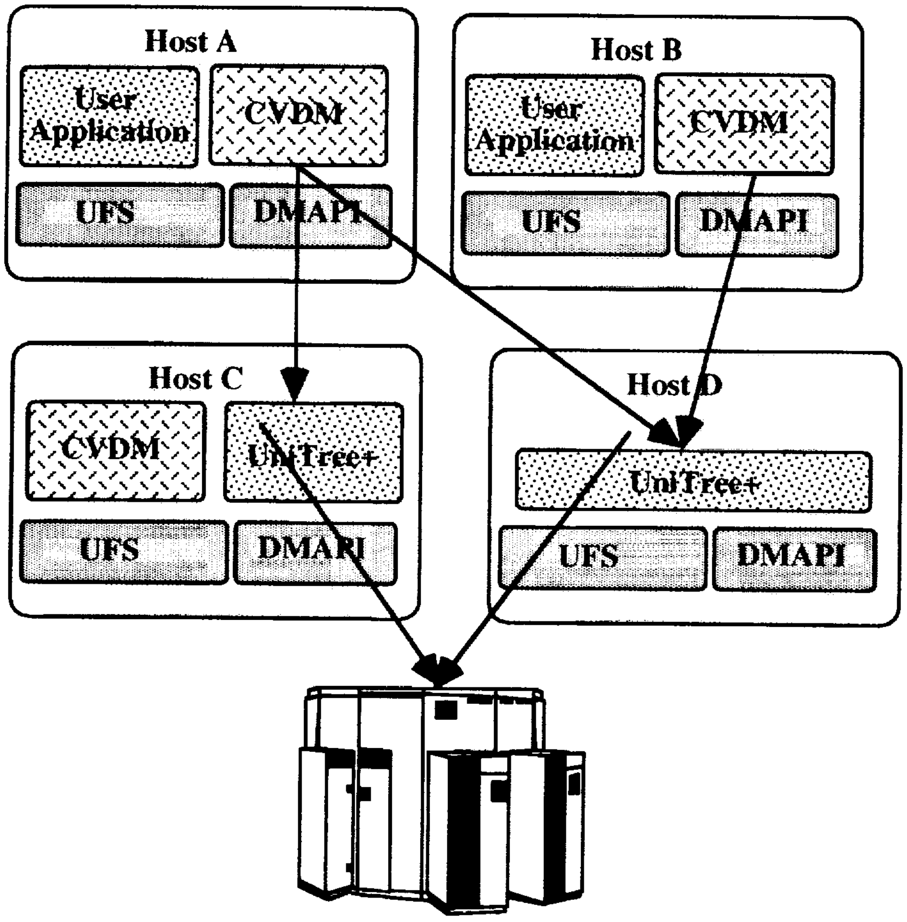

Figure 4: CVDM/UniTree+ Configuration Example

The above example configuration shows a configuration where the access servers are distributed over three distinct server platforms. The archive servers are also distributed over two distinct server platforms. This allows the site to tailor their system to meet the load that their environment places on the servers. If the system tends to be a read mostly system, increasing the number of access servers improves performance of the data retrieval. If a site uses their system in a write mostly environment, increasing the number of UniTree+ archive server platforms allows a system to handle very large data ingestion rates.

The architecture of a system running CVDM and UniTree+ on the same system appears as follows:

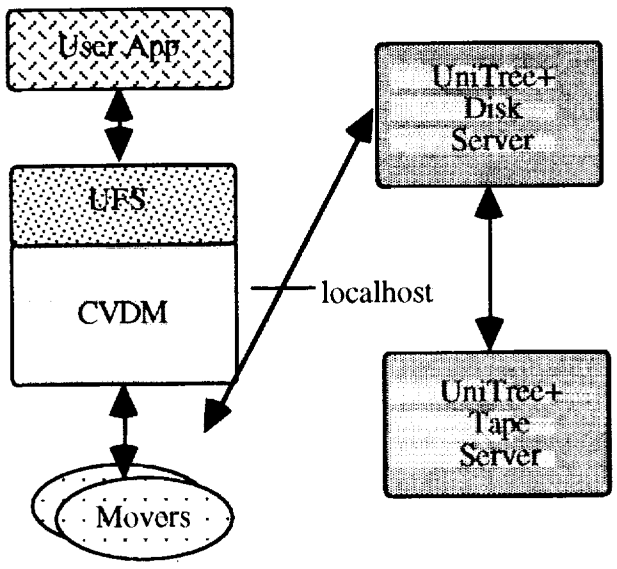

Figure 5: CVDM/UniTree+ Architecture

As shown above, the user application accesses the file data exactly as it would in a non HSM environment. Performance on resident files has shown to have < 1% deviation from native file systems not under HSM control. Except for delays for non resident files, the HSM is completely transparent to the application.

Migration in the current system transfers data from the native file system, via CVDM movers, to the UniTree+ disk cache, via UniTree+ disk movers. The data in the disk cache is marked to purge immediately and is therefore scheduled to move to tape from the disk cache in the next UniTree+ migration round. Requiring files to move from native disk to the UniTree+ disk cache has benefits, but also introduces some problems. The benefit of caching data is it allows remote CVDM servers to transfer data at network speeds. Data then is written to tape at tape speeds. The disk cache acts like a rate matching buffer between the network and tape devices. Tape devices are never tied up waiting on network transfers. The drawback is that the UniTree+ disk cache must be large enough to hold the largest file being migrated. Moreover, for local CVDM servers, there is no need for the rate matching buffer because the data can come from local disk. Non of these issues are architectural in nature, but instead are a result of merging CVDM and UniTree+ with extremely minimal changes to the UniTree+ base. In fact, UniTree+ can support both CVDM access servers and the existing UniTree+ access servers simultaneously.

Migration rounds in CVDM can be initiated in a variety of ways. Administrators can start a migration round via cron jobs or manually. Candidate selection is also configurable. A utility, migfind, scans the UFS name space looking for possible candidates. Weights are assigned to file attributes like size, modification time, creation time, owner, etc. Migfind then generates a sorted list of candidates to hand to migout, which in turn initiates migration. Migration rounds can also be initiated when space in a file system crosses preset thresholds. In this case, the system will first look for files that are migrated, but whose data is still cached in the file system. For these files, space can be freed by punching a hole in the file with the dm_punch_hole() call. If enough space is not freed by pruning cached data, a full migration round is initiated.

5.3 CVDM Futures

In future releases of CVDM, the UniTree+ tape mover and the CVDM disk mover will be collapsed into a unified mover for local CVDM servers. This unified mover will only be used for the local CVDM case so the rate matching nature of the UniTree+ disk cache will be retained for the remote servers. The use of a unified mover will dramatically decrease the overhead associated with migration from local CVDM servers.

Remote CVDM servers benefit greatly from buffering data in the UniTree+ disk cache. For large files, it may desirable to start the migration of data from the UniTree+ disk cache to the tape archive prior to completely transferring all of the file data. This is especially true for high bandwidth networks like HIPPI which can transfer data at rates above existing tape transports, or FDDI combined with low transfer rate tape drives. The UniTree+ system will be extended to support this.

6.0 Summary

The DMAPI has the potential to increase both the availability and quality of HSM products while providing functionality only available in the kernel intrusive implementations of today. As is usually the case, availability of the DMAPI will be driven by the market place. If customers ask for it, or as competitors begin winning sales because they have it, more OS and HSM vendors will deliver products based on it.

[1] Information in this section was obtained from the Data Migration Interface Group - Interface Specification. Version 2.0. This document is available via anonymous FTP at internet address 143.127.0.2