Pursuit of a Scalable High Performance Multi-Petabyte Database

Andrew Hanushevsky
Stanford Linear Accelerator Center
Marcin Nowak
CERN

Abstract

When the BaBar experiment at the Stanford Linear Accelerator Center starts in April 1999, it will generate approximately 200TB/year of data at a rate of 10MB/sec for 10 years. A mere six years later, CERN, the European Laboratory for Particle Physics, will start an experiment whose data storage requirements are two orders of magnitude larger.

In both experiments, all of the data will reside in Objectivity databases accessible via the Advanced Multi-threaded Server (AMS). The quantity and rate at which the data is produced requires the use of a high performance hierarchical mass storage system in place of a standard Unix file system. Furthermore, the distributed nature of the experiment, involving scientists from 80 Institutions in 10 countries, also requires an extended security infrastructure not commonly found in standard Unix file systems.

The combination of challenges that must be overcome in order to effectively deal with a multi-petabyte object oriented database is substantial. Our particular approach marries an optimized Unix file system with an industrial-strength Mass Storage System. This paper describes what we had to do to create a robust and uniform system based on these components.

Introduction

The BaBar experiment at the Stanford Linear Accelerator Center, SLAC (http://www.slac.stanford.edu/), is designed to perform a high precision investigation of the decays of the B-meson produced from electron-positron interactions. When the experiment starts in April 1999, it will generate approximately 200TB/year of data at a rate of 10MB/sec for 10 years. Specifically, 100,000 object oriented 2GB-sized databases are created each year. Once created, the data are rarely updated and serve as a reference for further analysis. The quantity and rate at which the data are produced, let alone analyzed, requires the use of a high performance scalable storage system. Constructing such a system, unfortunately, still remains elusive.

While hardware has progressed with higher media densities that grow at an almost exponential rate, transfer rates increase almost linearly. This quickly gives rise to the ability to store increasing amounts of data with an effectively decreasing amount of time in which to process it. Software solutions to this dilemma have been devised. For instance, IBM's High Performance Storage System, HPSS (see Figure 1 or refer to http://www5.clearlake.ibm.com:6001/), allows striped (i.e., parallel) point-to-point data transfer for any supported media. However, when such solutions are applied to tape media, they become overly complex, troublesome to administer, have limited error recovery, and are costly to scale. These problems practically limit tape transfer rates to single controller speeds. Thus, the Babar experiment is not capacity challenged, it is speed challenged.

This situation becomes even more problematic as larger experiments are being planned. For instance, CERN (http://www.cern.ch/), the European Laboratory for Particle Physics, is currently building a new accelerator, the Large Hadron Collider (LHC). A number of physics experiments at the LHC are scheduled to start taking production data in 2005. The data taking period is expected to last 15 or more years, with data rates ranging from 100MB to 1.5GB per second, depending on the experiment. The total collected data sample will be in 100PB range.

Improving Access Time

Our approach to solving this problem is to implement an architecture that can:

1. store unlimited amount of data,
2. optimally transfer large and small blocks of data,
3. dynamically load balance among n-servers, and
4. replicate databases on demand.

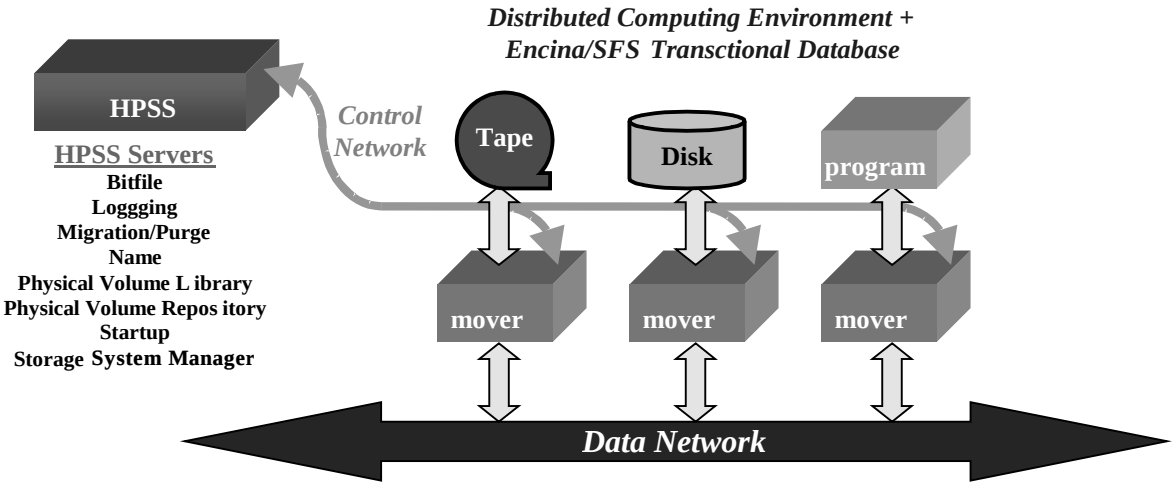

Figure 1: ***High Performance Storage System***

Unlimited Data Storage

The notion of storing unlimited amount of data is not new. Many Mass Storage Systems are available to do this. We chose IBM's High Performance Storage System (HPSS), an industrial strength system using a secure infrastructure and a fully transactional database for storing all system meta-data to prevent any undetected loss of data. This is critical for experiments such as ours where the data can never be regenerated.

HPSS is also unique in that it provides point-to-point data transfer between a source and a sink which can be a device or an application, as pictured in figure 1. The main advantage of such an architecture is the minimization of the number of nodes that exist between the source and sink; thus maximizing the transfer rate. However, because the control functions have been separated from the data transfer functions, it can be expected that there is a substantial latency in starting any individual data transfer operation. Indeed, this is the case with HPSS which excels with large transfer units (>100KB) but suffers with small transfer units (e.g., <100KB), as shown in figure 2.

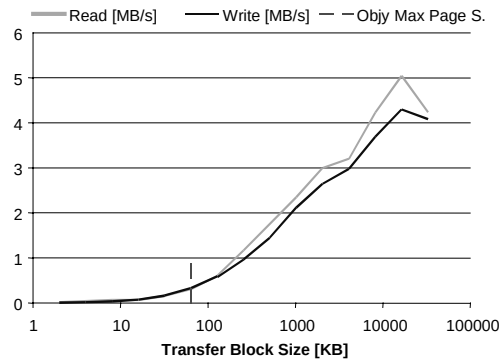

Figure 2: HPSS Simple API Performance

Database Access with HPSS

The Objectivity database employs a specialized data server daemon (AMS) that reads database pages on remote servers and transfers them to the client application. The AMS uses a set of standard POSIX.1 filesystem I/O routines to perform operations on directories (open, read, close) and database files (create, open, read, write, truncate, rename, etc.). Hence, the AMS assumes that it has access to a standard POSIX-compliant filesystem.

On the other side, HPSS offers several user interfaces, out of which the following could be used by an ODBMS: C language API, FTP/PFTP (parallel FTP) and NFS V2. However, the main purpose of the NFS interface is to provide access to HPSS name space and bitfile data and is not efficient for data transfer. This leaves the C language API and PFTP as the candidates for integrating Objectivity/DB and HPSS.

One possibility would be to provide HPSS access via the filesystem's vnode layer. We rejected this option because:

1. It would make us sensitive to operating system internals,
2. It would not solve the poor small block transfer time, and
3. It would not solve remote procedure call timeout problems when a file had to be brought back from tape.

A comprehensive solution would require that AMS provide dedicated support to a MSS as well as provide support to enhance transfer rates. Consequently, we approached Objectivity, Inc to provide a customized AMS. Together, we provided MSS support by:

1. exposing the internal filesystem layer[1],
2. load balancing protocol,
3. RPC timeout extension protocol,
4. Usage hints protocol, and
5. Security protocol.

While security is not performance related, it is an often overlooked aspect whose lack renders any performance improvements moot.

The ability to expose the filesystem layer allowed us to link any arbitrary filesystem, including HPSS, with the AMS, as shown in figure 3 where the oofs layer glues the AMS to HPSS.

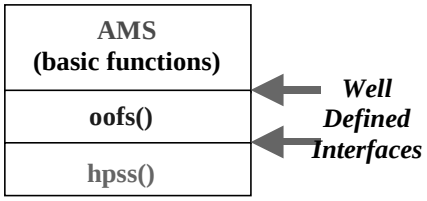

Figure 3: The layered AMS

However, figure 2 shows that the optimal data transfer size of HPSS does not match that of Objectivity/DB. The database requests data in pages or groups of pages, but a single transfer does not exceed 64KB (the vertical line in figure 2). On the other hand, HPSS achieves best performance when transferring megabytes of data in a single request or when utilizing data streaming techniques. The lower performance for small data blocks can be explained by the overhead associated with every read and write request, when different HPSS components need to be contacted. For example, a single I/O request involves the Bitfile Server, Storage Sever and than a Data Mover, which finally opens a dedicated network connection to the client and does the actual data transfer. In case of data streaming the connection is kept open all the time and there is no extra overhead.

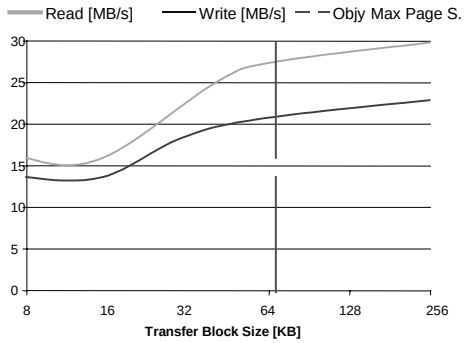

Figure 4: Veritas filesystem performance

In fact, we considered providing special code in the oofs layer in order to group I/O request and perform larger data transfers. However, the pattern of read and write calls executed by AMS is closer to random access then to sequential, which makes the implementation difficult and does not guarantee significantly improved performance. Also, high performance for small block transfers is readily achievable with certain file systems, as shown in figure 4 for a 5 disk RAID 5 configuration with the Veritas filesystem (http://www.veritas.com/).

Our solution to this problem was, in some ways, quite obvious. We simply married a file system whose transfer characteristics were optimized for small block transfers yet was capable of handling large blocks with HPSS, as shown in figure 5. The ooss (Objectivity Oriented Staging System) layer provides access to files in the local filesystem, in the same way as the standard AMS does. But if the file is not found in the local filesystem, it is copied to the local filesystem (i.e., staged) from HPSS.

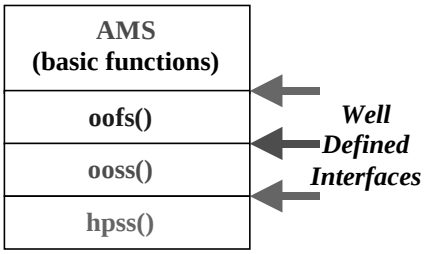

Figure 5: Extending oofs/HPSS

Since the ooss layer effectively glues a native filesystem with HPSS, it is in the best position to optimize the use of each system. It is also the natural place to implement algorithmic oriented changes that can substantially improve performance.

Dynamic Load Balancing

Dynamic load balancing has two components in out design:

1. Directing clients to the least loaded server and
2. Creating more data access paths via replication, as needed.

These mechanisms are dynamic because client-server associations and replication decisions are made in real time based on currently available load information and policy.

[1] Known as the Objectivity Open File System (oofs) interface.

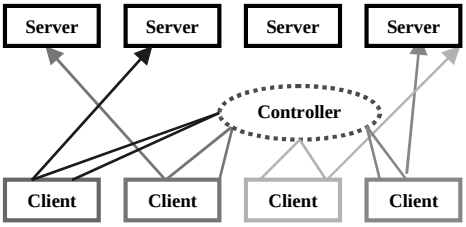

Figure 6: ***Dynamic Load Balancing***

Dynamic client-server associations are pictured in figure 6. The ability to direct a client to an arbitrary server not only provides a load-balancing mechanism, it also automatically increases the number of data transfer paths as the load increases.

The mechanism to implement dynamic load balancing is profoundly simple. I/O requests are directed to a control server (i.e., controller). The controller merely redirects the request to the least loaded server. Future I/O requests continue to be sent to the chosen server until that server redirects the request. Redirection can be used to balance the load using one or more of the following criteria, among others:

- Number of clients,
- Available real memory,
- Available disk space,
- Network link performance, and
- Service level agreements.

However, redirection implicitly assumes that each server contains an exact copy of all the data. Given the amount of data involved, this would not be practical. Instead, the controller redirects requests only to servers that are known to contain the requested data. This is usually a subset of all available servers. Should the load increase beyond the capacity of the current set of servers for some piece of data, the controller automatically orchestrates a replication operation allowing the load to be spread across an additional server, effectively providing a larger pipe to the data. When the load falls below a certain threshold, the replica can be eliminated to reduce disk space contention. Thus, redirection can also be used to temporarily increase transfer rate capacity when the need arises.

Dynamic replication can be easily implemented because HPSS provides a common repository for all of the data. The mere act of directing a request to a database server has the side effect of replicating the data associated with that request. This is because the ooss layer automatically stages the data if it is not present in the local file system. This greatly simplifies the Controller's bookkeeping and naturally distributes functionality. In order to maintain proper replicas, the Controller makes sure that all update requests are directed to the master read/write copy. When all updates are completed, that copy is made available in read-only form to potentially all other servers by simply migrating the database back into HPSS. This is possible because our experimental data is largely read-only and updates to the databases that can be replicated are strictly controlled.

Dealing with latency

High latency is a common part of all Mass Storage Designs that include offline storage (e.g., tape). Unfortunately, it is one least likely to be dealt with in systems that expect all of the data to be online. The usual problem is that the requestor times out waiting for the data and assumes that the request failed. The request may be retried, but more likely than not, the request will abort.

We dealt with the issue by designing a defer protocol into the AMS. For instance, when the ooss finds that it must copy data into the local filesystem from HPSS, it determines the length of time that it will take to make the file available and reports the time back to the client via the AMS. This allows the originating client to wait that amount of time and then re-issue the request. This eliminates one of the more vexing problems introduced by "back-door" migration schemes – client timeouts and retransmission storms. This mechanism is called the Deferred Request Protocol.

Client supplied usage hints

Another important feature that has been added is the ability to allow a client to send hints to the oofs as well as the ooss layers. These hints can be anything that the implementation can understand such as processing options (e.g., sequential or random), clustering and parallel-staging hints. Properly used hints can substantially reduce overall processing time. This mechanism is called the Opaque Information Protocol in that all information generated by the client is simply relayed without inspection or change to the layer that understands the information.

Security

Finally, given that the BaBar and LHC collaborations are wide-spread, security became a

critical requirement. It matters little if one minimizes processing time only to keep spending it to recreate hacked data. The AMS architecture allowed us to implement a general authentication scheme that can be based on most existing security protocols (e.g., Kerebros, PGP, etc.). The authentication scheme allows us to implement appropriate authorization levels to maintain the safety and integrity of the databases. The mecha nism is called Generic Authentication Protocol.

CERN Configuration

The initial intention was to use the HPSS system at CERN for managing both the tertiary storage and all associated disks. The disks would form the highest layer of the migration hierarchy, where files would not reside permanently but only when they are opened by an application. The AMS was to access the files via the POSIX compliant HPSS programming API.

After our experience with the first prototype of the interface, the decision was taken to remove a large part of the disks from the control of HPSS and let AMS access them directly. In such configuration, when client application tries to open a database, the database file is imported from tape and stored in the local disk pool. All subsequent I/O operations on this database file can be performed by AMS with the full speed of the local filesystem.

Of course, the control over the disk pool comes with all the associated disk space management issues. The interface must care for staging in files, migrating them back to tape and purging when there is not enough free space in the pool (hence the name: staging interface). For this purpose a new module has been added to the interface - migration daemon. The daemon periodically scans the disk pool and copies new or modified files into HPSS. When the amount of free disk space falls below a predefined threshold, the migrated files are deleted from the pool. File locking is used to synchronize operations of AMS and the migration daemon.

The staging interface used at CERN is linked with the Remote File I/O library (RFIO). RFIO is a package developed at CERN and used by the HEP community. Its main purpose is to provide all standard file I/O operations on remote files, with better performance than NFS V2 does and without directory mounting. RFIO effectively replaces the HPSS library in the interface. This allows deployment on all major UNIX platforms, regardless if they are supported by HPSS or DCE, required by HPSS, which is of importance at CERN with its heterogenous computing environment. The HPSS servers at CERN run the RFIO daemon, which translates RFIO requests into HPSS API calls and arranges for data transfer connections. RFIO makes full use of HPSS parallel, point-to-point data transfer capabilities, and no performance degradation has been observed. Figure 7 presents logical view of AMS/HPSS interface using RFIO to communicate with the server and initiate data transfers.

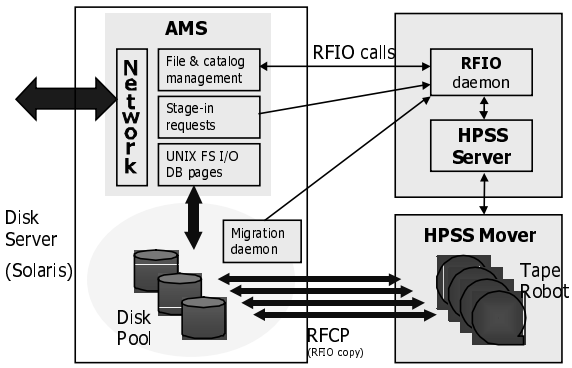

Figure 7: AMS *staging using* RFIO & HPSS

The terabyte database test

The first practical large-scale test of all the involved components has started in November 98. We are planning to store 1TB of data in object database format in HPSS using the described AMS/HPSS staging interface. The primary goal is to prove the functionality and capacity of the system, with performance as a secondary goal. The foreseen fill rate is between 1 and 2MB/s, which should allow to store 1TB in about 2 weeks. The configuration of the test system is described on Figure 8.

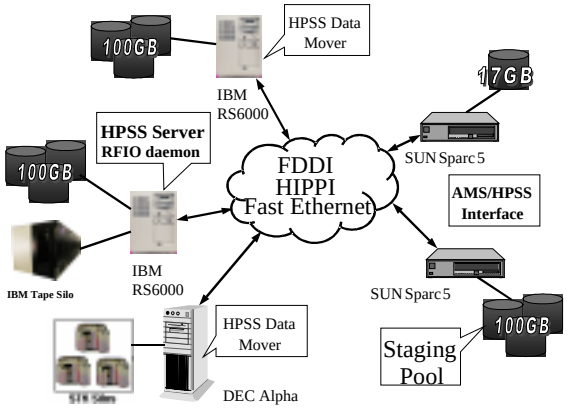

Figure 8: *Configuration for 1TB Database Test*

The experience gained during the 1TB test should allow CERN to move into the hundred-terabyte region in year 1999 and into the petabyte region later. More attention will be given to the data rates, which should grow to 35MB/s in 1999 and to 100MB/s and more in 2005. As both HPSS and Objectivity/DB has been separately shown to sustain over 30MB/s data rates, this task is certainly achievable.

SLAC Configuration

We approached the notion of having HPSS manage all aspects of the storage system with great skepticism. Based on previous experience, we knew that HPSS was unlikely to perform well with small transfer units. However, our first task was to implement the oofs and show that AMS could, in fact, interface with other complex filesystems. The initial proof of concept used the HPSS POSIX API as the filesystem.interface. As expected, this marriage of AMS and HPSS was a poor performer.

Our next step was to implement a system that used a fast local cache (i.e., Veritas filsystem) with HPSS providing tertiary storage (i.e., Redwood tapes). We decided to use pftp as the transfer protocol between the cache and HPSS. The decision was based on our need to quickly develop a working prototype. Figure 9 illustrates the architecture of this design.

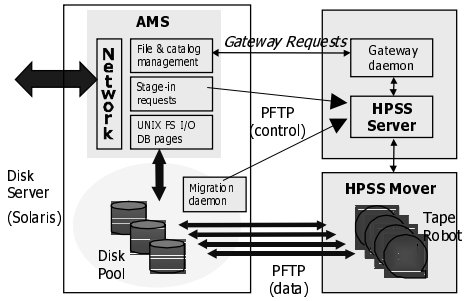

Figure 9: AMS *staging using* PFTP & HPSS

In the design, all AMS data operations go through the oofs layer. The migrating filesystem is implemented in the ooss layer which oofs uses. When a file is opened, if it does not exist in the local filesystem, ooss stages in the file from tape using pftp. A separate migration process works independently to maintain a target amount of free space in the local filesystem. Modified files are migrated back to tape and least recently used files are purged. Files can also be staged or migrated on demand and purged based on a particular time and date.

The system uses the HPSS Name Server as the authoritative file catalog. Indeed, the only files that exist in the local filesystem are those that are being actively used. This meant that we had to implement a special gateway process to issue HPSS commands from a remote host (i.e., the AMS). These commands are typical metadata oriented operations such as name lookup, rename, erase, etc.

Figure 10 shows our test configuration. It consists of a single AMS with a 900GB local cache and a single tape server to which two Redwood drives have been attached.

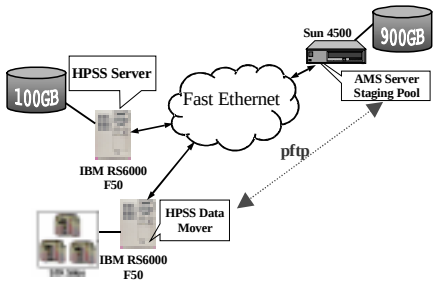

Figure 10: *Configuration for 1TB Database Test*

We used this configuration to create over 4,000 databases equaling approximately 4TB of data. What impressed us was the stability and balance of this configuration. We suffered no system-related failures and transfer rates were limited by network speed. Consequently, we could only achieve an aggregate transfer rate of approximately 9.8MB/Sec. Even at the maximum possible database creation rate, the migration process had little difficulty in keeping up with the system and maintaining adequate free space.

Our choice of pftp proved to be fortuitous. Since the protocol has been widely implemented, the system is essentially independent of the mass storage system being used. This allowed us to create similar configurations for other laboratories that did not have HPSS, as well as allowed us to be in a position to switch mass storage vendors without impacting the system.

Of course, we knew that the production system needed to be substantially faster to handle the expected 1 TB/day databases creation rate as well as the 1-2TB/day analysis rate (i.e., 3 TB of data moved per day). Our approach was to further distribute the hardware and upgrade the network. A production AMS would serve both as a database server as well as an HPSS data mover, albeit with a ftp-like interface for implementation independence. Figure 11 illustrates this architecture.

In the figure, each AMS server has two or more tape drives attached to it. Thus, when a stage-in or stage-out operation is needed, data can be transferred from local disk to local tape using shared memory instead of the network; providing the highest possible transfer rate. The system is also self-correcting. Since all of the

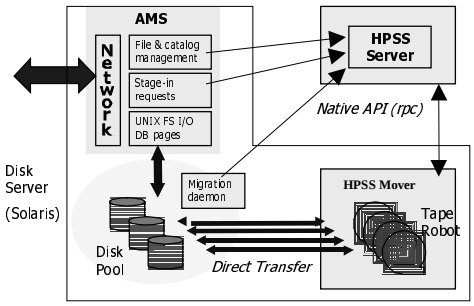

Figure 11: AMS *Staging with a local tapes*

tape drives are controlled by HPSS, should an AMS need more tape resources than it has locally available, additional tape drives, attached to other AMS hosts or perhaps dedicated tape servers, can be transparently used.

The system is also highly scalable since we can add as many of these AMS/Mover nodes as needed to handle the load. Dynamic load balancing allows us to easily spread the load across multiple hosts as well as recover from failed hosts. Our production configuration for the first year of operation is shown in Figure 12.

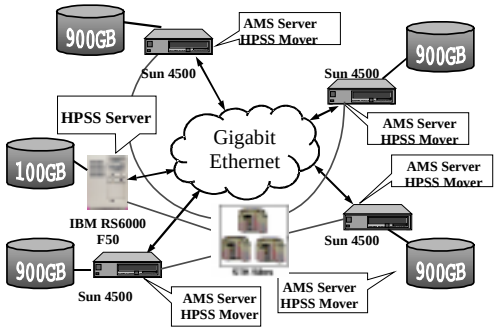

Figure 12: *1st Year Production Configuration*

In this configuration, there are four AMS/Mover hosts providing approximately 4TB of disk cache. These hosts can communicate with each other as well as their clients through gigabit switched ethernet. This configuration provides maximum flexibility, scalability, and performance achievable at modest cost increments. Consequently, we feel that we have finally addressed the BaBar speed challenge.

References

[1]. IEEE Storage System Standards Working Group (SSSWG) Project 1244, "Reference Model for Open Storage Systems Interconnection, Mass Storage Reference Model Version 5", Sept. 1994.

[2] Jamie Shiers, “Building a Database for the LHC – the Exabyte Challenge”, Proceedings of the 15th IEEE Symposium on Mass Storage Systems, April 1996.

[3] Jamie Shiers, “Massive-Scale Data Management using Standards-Based Solutions”, Proceedings of the 16th IEEE Symposium on Mass Storage Systems, September 1997.

[4] David Fisher, John Sobolewski, and Terrill Tyler, “Distributed Metadata Management in the High Performance Storage System”, Proceedings 1st IEEE Metadata Conference, April, 1996, see http://www.sdsc.edu/hpss/hpss1.html.

[5] R.W. Watson and R.A. Coyn,e “The Parallet I/O Architecture of the High Performance Storage System (HPSS)”, Proceedings 14th IEEE MSS Symposium, September, 1995.

[6] Objectivity, Inc., “Objectivity Technical Overview”, 1996.