Building the Mass Storage System at Jefferson Lab

Ian Bird, Bryan Hess, Andy Kowalski
SURA/Jefferson Lab
12000 Jefferson Avenue,
Newport News, VA 23606
Ian.Bird@jlab.org, Bryan.Hess@jlab.org, Andy.Kowalski@jlab.org
tel +1-757-269-6224
fax +1-757-269-6248

Abstract

Thomas Jefferson National Accelerator Facility (Jefferson Lab) is a U.S. Department of Energy Facility [1] conducting Nuclear Physics experiments that currently have data collection rates of up to 20 MB/second. Future experiments, however, are expected to greatly exceed these rates. Post processing and data analysis produce similar amounts of data. Both the raw and processed data are stored on tape and need to be easily accessible. Between data collection and processing, the Mass Storage System at Jefferson Lab currently moves over 2 TB of data per day.

This paper describes the requirements, design, and implementation of the Jefferson Lab Asynchronous Storage Manager (JASMine), developed to support multi-terabyte per day I/O. JASMine is a lightweight, distributed, scalable, high performance, high capacity system. It is built around a relational database and is implemented almost entirely in Java. As a result, any system that supports a Java Virtual Machine can run JASMine, or JASMine services, with few if any code changes.

1 Introduction

For the past few years Jefferson Lab has made use of Open Storage Manager (OSM), a commercial Hierarchical Storage Manager from Computer Associates to manage data stored on tape. OSM, however, is no longer supported and has many shortcomings that limit its performance and scalability. Since it is not a distributed system, all the tape drives are required to be connected to a single server, making the server an I/O bottleneck.

To work around these shortcomings, we have developed a front end and user interface. The front end, called TapeServer, provides disk cache, cache managers, data staging, smart scheduling, and multiple OSM server support. The disk caching provided by the TapeServer improves the I/O throughput of the overall system. Smart scheduling provides a way to prioritize and limit user requests. Making multiple OSM servers appear as a single system to the user helps with the centralization of metadata as well as the ability to add additional tape drives on new systems. Multiple instances of OSM, however, require additional administration.

It has become essential to replace the functionality of OSM within the TapeServer software. Although other Mass Storage Systems and Hierarchical Storage Managers exist, none simply fit into the existing infrastructure and provide the needed functionality. A lot of the required functionality (disk-cache managers, data staging, and smart

scheduling) was already in place in the TapeServer software with OSM simply being used to manage the metadata and manipulate tapes. The replacement of OSM within the context of the existing TapeServer software was thus a relatively simple and straightforward project.

2 Requirements

JASMine must provide a means to manage data stored in tape libraries, tools for reading and writing those tapes, and user commands to interact with the system. All persistent metadata must be stored in a reliable and reasonably fast database. This includes both information about tape contents and user requests. Simple user commands are needed to read and write data files to tape from various systems connected to the Jefferson Lab network. Additional user commands are needed to retrieve data files from tape and deposit them on automatically managed file servers - the cache servers.

2.1 Automated (Lights Out)

The Jefferson Lab Computer Center is not staffed 24-hours a day. Therefore, the Mass Storage System must be able to work without operators. It needs to be completely automated and capable of detecting and disabling suspect hardware components. It must be able to continue to function in a degraded mode until staff members arrive and correct problems such as drive failures.

2.2 Hardware

The more distributed the system, the better the performance. PCs running Linux will be used for the servers and data movers, but the software will run on any system with a modern Java virtual machine.

A StorageTek PowderHorn Tape Library is already in use at Jefferson Lab. The software must work with this tape library and its attached tape drives. These currently include StorageTek SD-3 (Redwood) and 9840 drives, with 9940 drives expected in 2001.

2.3 Disk Cache

Many of the client systems requesting data from tape may not be located on fast or well managed networks and are thus unable to keep up with the I/O rates of high performance tape drives. In order to optimize the utilization of the tape drives, local disk caches of RAID subsystems are used.

In addition, user requests are not guaranteed to be unique. There will be times when multiple users will be requesting the same file or files. Disk caches will minimize tape accesses and increase throughput by reading the file from tape once and using the copy on the cache disks to fulfill all the requests.

2.4 Distributed Data Movers

The overall throughput performance of the system must be easily scaleable. The software needs to be designed such that the overall performance of the system increases as additional data movers are added. Previous systems such as OSM required all tape drives to be connected to the same server as the database. This created unavoidable server

bottlenecks including network interfaces, server IO boards, processor power, and disk IO. To eliminate such bottlenecks, the system must consist of distributed data movers.

Distributed data movers will also eliminate many single points of failure. A crashed data mover will simply degrade the overall performance of the system instead of halting it all together.

2.5 Smart Scheduling

Access to data cannot always be granted on a first come first served basis. Because tape mounts take time and reduce the overall throughput of the system, requests for files on the same tape should be processed consecutively, while the tape is mounted. While this will help maximize throughput, it could also lead to job starvation. Therefore, the system must be able to prevent job starvation by tracking usage by users and hosts. Requests should first be prioritized by usage and then by available resources.

2.6 Monitoring and Management

System administrators and users need tools to allow them to monitor and react to the status of the system. Values monitored should include the status of each server and service, the data rates of each server, resource utilization, and the prediction of request execution.

The status of a server or service is either available or not, allowing administrators to identify downed services easily. The data rates of each service that moves data will show the throughput of the system as well as identify bottlenecks. Resource utilization will also identify bottlenecks caused by insufficient amounts of resources such as disk space and tape drives. The estimates of request execution time will provide users with an idea on when they can expect their request to begin execution. This information will also be used by external systems requiring estimates for the time required to access data. Such systems could include Grid applications.

In addition to the statistics provided by monitoring, the system must provide a way to generate statistics from history files or logs. Such information will include the number of requests during a given period of time, the amount of data moved, and the availability of services and hardware components.

Finally, a set of management tools will include the ability to add tapes to the tape library, eject tapes from the tape library, enable/disable data movers, enable/disable cache managers, and manipulate user requests.

2.7 Error/Exception Handling and Recovery

When an error occurs, the system needs to provide an informative description of the error and execute an appropriate action to work around the problem. Such action may include retrying actions, disabling services, disabling the use of certain hardware components, or alerting administrators of problems.

In the case of a database failure, the system needs to be easily recovered. Procedures for database and metadata backup and recovery will be required. The metadata must be

stored in at least two separate locations, including the database, such that one copy can be used to update the other after a failure. The loss of log or statistical information is acceptable. In the case of metadata, nothing should be lost in the event of a failure.

2.8 Reservation of Resources

To ensure the availability of resources for a request once it has been started, resource reservation is essential. A request needs to reserve the disk space required for caching or staging the data. A request also needs to reserve a tape drive for use.

3 Architecture

The JASMine system is designed using object-oriented software engineering principles. Initial design iterations used interactive UML modeling tools. From the UML designs prototype Java package and class structures were created. The quick design cycle from UML to code and back encouraged good object oriented design that remained tied to practical concerns like the importance of high performance, and error handling. Pragmatic development practices such as those described in [2] also provided insight into the construction and testing of JASMine.

Because JASMine is a large system, many small prototype implementations of critical or complex components were constructed and tested, with the results fed back into the overall design. This led to the development of tested, reusable components and good internal API development. In this way we demonstrated the viability of JASMine's design: Our component-level concerns were satisfied by prototyping and our high-level design issues such as error recovery and scalability were resolved by UML analysis and lessons learned from our previous experience with OSM and the layered TapeServer software.

JASMine is almost completely implemented in Java. Certain low-level library and tape interface routines were coded in C using the Java Native Interface [3]. This allowed us to make aggressive use of the object model for our key architectural features: The storage object, the management of distributed components, the efficient movement of data, error handling, and issues of scale.

JASMine is comprised of an SQL database, distributed data movers, distributed disk-cache managers, and a number of administrative daemons. JASMine is designed to have configurable request scheduling and few single points of failure. The database contains all file information, system state information, user requests, and statistical information. Robustness in the administrative daemons is achieved by replication of services. The data mover is designed so that individual machine failures will merely degrade overall performance.

3.1 Overview

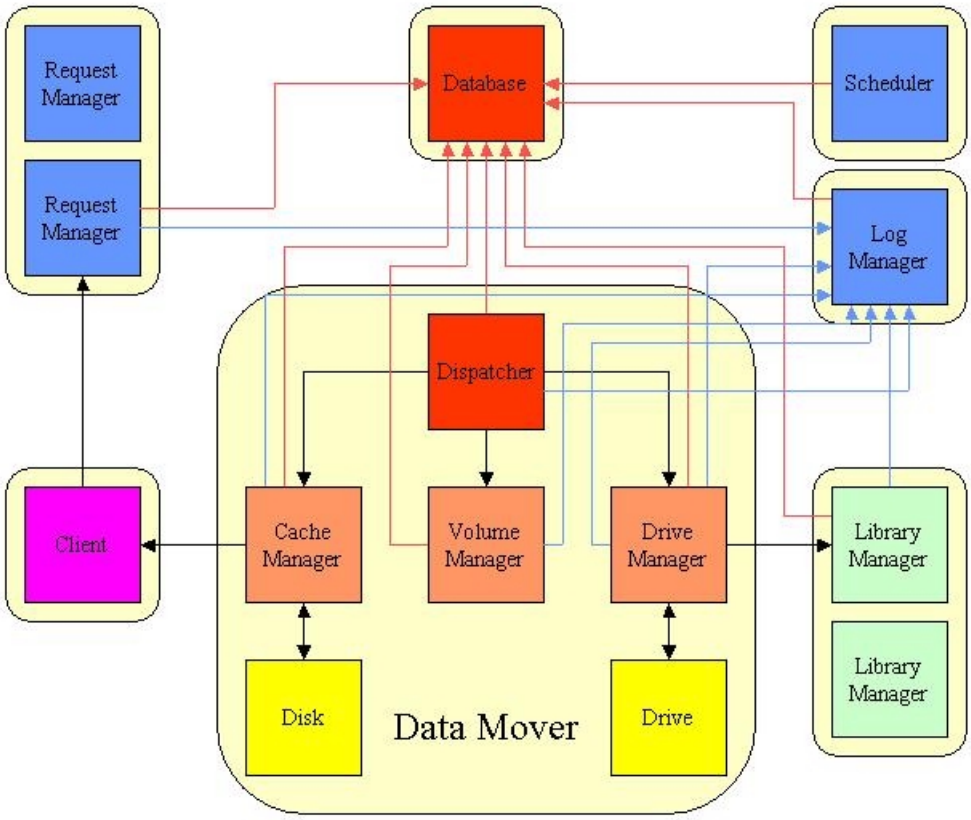

Figure 1.

3.2 Storage Objects

Storage objects fall loosely into two categories: The logical and the physical. Logical components are organizational constructs while physical components have a concrete aspect. A Storage object like a tape cartridge has both physical and logical attributes. The location, type of cassette, and status of the write-protect switch are physical attributes. The volume group to which a volume belongs and the file metadata contained on the volume are logical constructs. The union of all the attributes, both physical and logical, is the largest unit in the System, the Store. A store contains tape libraries, drives, volumes, bitfiles, and other items.

In figure 2 we show the inheritance diagram for Store objects. It may not be immediately clear why we would establish what is normally taught as an "is-a" relationship between, say, a BitFile and a Volume. One would not at first glance think that a bitfile "is a" volume, but consider a second interpretation of the inheritance model. A bitfile has a need for a superset of the attributes and operations that a volume has, and a bitfile always resides on a volume. It makes sense then to query a BitFile object for its volume number, or to ask if it is currently loaded in some tape drive.

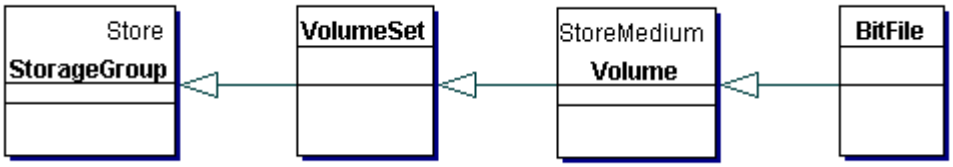

Figure 2.

This style of relationship becomes powerful when it is wed to a back-end database. Each level in the inheritance tree queries a different table. There are four basic database interactions, each of which is handled in a slightly different way:

(1) Database inserts - new store objects are created using static factory methods like *BitFile.newBitFile()*.

(2) Database queries are made by non-static instance methods in a lazy way: when an object is constructed the store object name is recorded in the object, but no database query is performed. Part of each store object's internal state includes a flag to indicate if it needs to query the database. Every access method calls a private *refresh()* method that will contact the database and load all the items from one table if the data is not current.

(3) Joins are mostly performed by methods called from constructors with names starting with "lookup". This is the most complex part of the object model because of the way in which it uses polymorphism. Each store object defines a lookup method that takes a storage object name as a string and returns the associated store object in the current object's table. That is, *Volume.lookupVolumeName("DLT398")* will return DLT398, an obvious no-op. This method is overridden in BitFile, however, *BitFile.lookupVolumeName("2000.10.09.02.03.04.049.mss1.jlab.org.345483745")* will also return DLT398 if DLT398 is the volume containing that bitfile name.

When the constructor for BitFile calls its super-constructor, it only knows its bitfile name and it MUST call the super-constructor before it does anything else. The super constructor will call its super-constructors (which we ignore for the moment) and then it will set its name by calling *lookupVolumeName*. If the object being constructed is ultimately a BitFile and not just a volume, then this call will be dynamically bound to the overridden implementation of *lookupVolumeName* in the BitFile class. The overridden method does a join between the BitFile and Volume tables to find the volume that the bitfile belongs to.

(4) Database updates are made by non-static instance methods. For a BitFile b, *b.setATime()* will update the access time to the current timestamp. All updates are write-through. That is, they are immediately written to the database. All methods that alter the database also invalidate the object's cache of database information so that it will be reloaded the next time an accessor method is called. This helps to prevent unnecessary queries. An update does not necessarily mean that the object will be queried again.

This object hierarchy includes an important use of encapsulation to produce closure-like semantics for the database queries. The accessor and modifier methods in the store

classes are declared public and they make use of three privately declared methods, namely *refresh()*, *invalidate()*, and *needsLoad()* that are defined in every store-derived class. Because they are private they are not overridden in subclasses-- should a Bitfile object call *isMissing()*, a Volume method, this will in turn call the *refresh()* private method in Volume. Should the BitFile object then call *getWriteVols()*, a VolumeSet method, the *refresh()* that is private to VolumeSet will be called. If the *refresh()* method were not private then the overriding would break this implementation of closures. This is done so that we can preserve per-table loads from the database and maintain information for each table about its cache state even when the object contains caches of several database tables as a BitFile or Volume object must.

3.3 Authentication

The original TapeServer software required all clients to be part of Jefferson Lab's central computing environment and relied on host-based authentication. JASMine exchanges this for a framework of pluggable authenticator modules that establish identity on a network connection before it is returned to the application. Using these authenticators provides a low-level security API to establish identity before any other transfer takes place. For instance, off-site users could be required to authenticate with a username and password that are encrypted before transmission, but this rigor is not necessary for on-site transfers between persistent servers, they share authorization information through the database and need only a permissive authenticator for server-to-server traffic. Other mechanisms could involve ssh-keys, kerberos, or other single-sign on systems associated with Grid projects. This pluggable authentication scheme is applied to java RMI connections using custom socket factories and in peer-to-peer communication using the file moving protocol that we describe in section 3.5.1.

3.4 Tape Label and Data Formats

Like most tape storage systems, JASMine makes use of tape labels. Labels allow systems to verify that the correct tape has been mounted in a drive. Labels are also used to gather information required to utilize a given tape. Such information would include, but is not limited to, block size, data format, and file position.

JASMine makes use of ANSI X3.27-1987 standard labels [4]. The ANSI standard provides a standardized labeling scheme for the interchange of tapes between sites. Most tape storage systems make use of ANSI labels. Such systems can therefore make use of tapes created by JASMine.

The ANSI X3.27 standard, however, is old. As a result, certain fields in the 80-byte labels are not large enough to handle the values required by today's storage systems. For example, the StorageTek Redwood tapes drives are capable of using block sizes of 256 KB and cartridges capable of storing 50 GB of data. Such cartridges can easily hold more than 100,000 files. The block size and file number can thus be too large to fit in their respective fields in the ANSI label definition.

To get around these limitations JASMine specifies zeros in the fields that are too restrictive and stores the needed information in a hidden JASMine label located after the 80-byte ANSI label and in a database. The ANSI standard states in section 6.2.1:

> A label shall be a record that shall have a length of 80 bytes. Each label shall be recorded within the first or only 80 byte positions of a block. If the block contains any additional bytes, they shall be recorded with any desired bit combinations.

Since the block size used will always be larger than 80 bytes, the additional space can be used to store metadata specific to JASMine. This does not violate the ANSI standard because anything after the first 80 characters is ignored by the standard. Sufficient data is, however, stored within the ANSI labels to allow other systems to reconstruct the files on disk using only the ANSI labels and a given block size.

Although ANSI labels with hidden JASMine labels are the default and preferred way to label a tape, JASMine can make use of other tape label formats. To allow for essential backward compatibility with tapes previously written by OSM, JASMine has support for OSM label and data formats. Other formats can be easily added to JASMine by extending the generic Label interface.

JASMine does not format the data before it is written to tape. Data is written directly to tape in its binary form. This allows the data on tape to be easily read back by hand using tools such as Unix's dd command. However, the cpio format is also supported for backward compatibility with OSM. Other data formats can be easily added to JASMine by extending the generic DataFormat interface.

3.5 File Movement

Digital object movement is the most basic operation of any mass storage system. While I/O is block-oriented at its lowest level, the essential primitive presented by the storage system is the file that is moved from tape to disk or from tape across the network to a client. Unlike a random-access file system, such transfers treat files as indivisible units. The enforcement of all-or-nothing file moving semantics is maintained throughout JASMine. Other data integrity checks like CRC32 checksums are made within the file moving components to preserve the completeness of file transfers.

The two primary file-moving components are the TapeIO component that moves data to and from tape and the JP file moving protocol that moves data between any two non-tape objects.

By using the stream abstraction we created a file-moving infrastructure that is fast but not limited to any particular use. Some mass storage systems require that secondary storage (usually disk space) be used as a buffer between the tape and the network. As speed ratios between tape drives and networks vary with changes in technology this staging alternates between being an asset and being a liability. Our solution is to move files using streams in such a way that we can either connect the tape stream to a stage disk object or directly to a client over the network. Particular attention is given to performance in our stream classes. Memory-to-memory data copies are kept to a bare

minimum and a fast, thread-safe double buffer is used to avoid starvation of tape drives under system load.

There are other cases where the stream abstraction proves useful. Within the data mover, we pass streams through tape formatter objects that completely define the layout of a tape. If we need to read an OSM tape or an HPSS tape, we merely choose the correct tape formatter connecting tape I/O streams to one end and our job to the other. The tape label is part of the formatter object. This is the mechanism used to implement ANSI tape labels.

3.5.1 Protocol

There is extensive code reuse in the file-moving portions of the system. This is achieved by defining a simple but extensible protocol for file movement. JP, the JASMine file moving protocol, uses Java serialized objects as messages passed over streams. The protocol is strictly synchronous to avoid complexity. Interaction follows a basic message/acknowledgement scheme. When asynchrony is necessary additional connections and protocol instances are created. Multithreading is used to keep the protocol simple while allowing for multiple outstanding requests between servers. For performance reasons our implementation of JP always falls back to a simple raw data transfer over tcp when bulk data transfers are made. That is, Java serialized objects are only used for metadata transmission.

JP includes hooks that allow its behavior to be tightly controlled in three ways. First, an Authenticator interface is defined to allow for pluggable authentication. Secondly, a pluggable file transfer policy mechanism may be defined to guarantee expected behavior. As an example, a request only meant to download files will be blocked by policy from uploading or deleting files. The final means for altering the protocol includes a hook for a message dispatcher. This hook allows for protocol extension, and is the way we reuse the File Moving code to create Cache Servers, Stage Disks, and interactive FTP-like file transfer agents. New messages with hooks into the appropriate database(s) are created and the whole of the JP code is reused.

3.6 Metadata

Metadata is important to both the user and the system. Users need metadata to determine which file they want and what resources are required to obtain the file. The system uses metadata to determine which tape the file is on and where the file is located on tape. Therefore, the metadata needs to be accessible without having to read the tape. Three copies of the metadata are created. They are stored in a database table, on tape along with the file, and in a flat file, called a stub file, that represents the file to the user.

The metadata consists of the following information:

1. Bit File Identification (Unique string of alpha characters and numbers).
2. Original full path file name.
3. Owner.
4. Group.
5. Permissions.

6. Size in bytes of the file.
7. Creation date and time of the file.
8. Volume label of the tape on which the file is located.
9. Position of the file on tape.
10. Format used to store the file on tape.

The database updates or inserts are made before the stub file is created. Therefore, the database is to be considered the most accurate. User access to the metadata, however, needs to be easier and more user friendly than a database query. So users access a directory structure on an NFS or CIFS share that contains stub files full of metadata. Each stub file represents a single file on tape. They are ordered in directories specified by the user. This gives the user the final say on the grouping of files and does not require any additional commands to browse the list. Users can use the standard commands available with the operating system to list the stub files in a directory.

Although the stub files currently exist on file systems shared via NFS or CIFS, there is no reason why this cannot be replaced later with a database or file system front end to a database for scalability.

Since there are three copies of the metadata, there are multiple ways to recreate metadata that is lost to an accident. Stub files can be recreated from the database. The stub files can likewise be used to recreate the database. Finally, the tapes can be read to recreate the metadata for either the database or stub files. The metadata stored on tape also helps sites determine what is on an exported tape or to generate stub files at their site with a given tape.

3.7 Database

JASMine uses Java Database Connectivity (JDBC) to connect to a MySQL database. This gives JASMine the flexibility to make use of any SQL database that supports JDBC connectivity with little or no code changes. Most, if not all, major SQL databases vendors have JDBC drivers. Previous testing had actually been done using the OpenIngres database (Computer Associates). Changing databases requires only installing a new JDBC driver.

MySQL is an Open Source relational database that supports the ANSI SQL standard (ANSI X3.135-1992). JASMine makes use of mostly ANSI specified SQL statements to limit its dependency on a single database solution.

The database tables are fairly straightforward. The columns defined in the tables correspond to the variables in the object they represent. The relationship between the tables reflects that of the objects they represent with relation to their super class.

3.8 Request Manager

The request manager is a replicable object that is the first point of contact for user-level commands. The request manager's function is to add requests to the database queue with a default priority and to return request identifier information to the user. The request

manager also returns informational requests that can be immediately satisfied without queuing; such as job status and system utilization statistics.

For queued requests the scheduler will order them in the database according to some policy, and dispatcher threads will select jobs to run. Subsequent communication with the user commands will come from the data mover(s) that turn the requests into jobs. The request manager will not be contacted again after the initial query.

3.9 Scheduler

The scheduler is hierarchical. Groups of hosts or users are assigned a single share. These groups are then divided into subgroups that are assigned shares for that level. Since the share of the parent group is checked first, the shares assigned to subgroups are effectively sub-shares of the parent group's share. The result is a hierarchical share tree.

The shares assigned can be dynamically changed so that resources can be reallocated as needed. To ensure that requests from certain hosts, like batch farm hosts, are started promptly, host groups can be created and assigned large share values. This will prevent farm hosts from sitting idle while waiting for data from the tape library. Smaller share values can be assigned to groups of desktop systems. Similar actions can be taken by changing user shares.

Two share trees exist. There is one share tree for hosts and one for users. The scheduler assigns priorities to hosts to prevent the overuse of the tape library by remote mini-farms and desktop systems for which the network bandwidth and system efficiency is limited or not known. Overuse by inefficient systems would have a negative impact on the Jefferson Lab batch and analysis farms. Priorities are also calculated for the users to prevent resource hogging by one or more individuals.

Each group, host, and user is assigned a priority. This priority is calculated by using the following formula:

```
priority = share  / ( .01 + (num_a * ACTIVE_WEIGHT) +
        (num_c * COMPLETED_WEIGHT) )
```

where ACTIVE_WEIGHT, and COMPLETED_WEIGHT are configurable parameters. The value of share is static and assigned to each group, host, or user. The number of active requests is represented by *num_a*. The number of completed requests is represented by *num_c* and is calculated for a given time period. The priorities represent the desired scheduling order for requests from hosts and users based on assigned shares and usage during that time.

When deciding on the ordering of the requests, the priorities of the host groups are first considered. This means that a low priority user will have their request dispatched before a high priority user if the submitting host for the low priority user has a higher priority than the submitting host used by the high priority user. This is to ensure that individual user requests from hosts throughout the Jefferson Lab network are not taking data resources away from the batch and analysis farms.

3.10 Cache Manager

A cache server is a large pool of disk space that is automatically managed by JASMine. As mentioned in section 3.5.1 we extend JP to include database hooks for cache disk management. The database is then shared between numerous cache servers to establish a high-throughput disk pool.

Along with the cache server there is a small cache client that provides an API for JP messages and file movement. A cache client may request, for example, that it be allowed to transfer a file from a cache disk. This involves the following interaction:

(1) Client C sends Locate (filename) to any cache server, S1.
(2) S1 decodes the message and looks up the file location in the database
(3) S1 packages and sends a reply message to C which states that the file in question is on some cache server S3.
(4) C contacts S3 and initiates a direct transfer of the file.

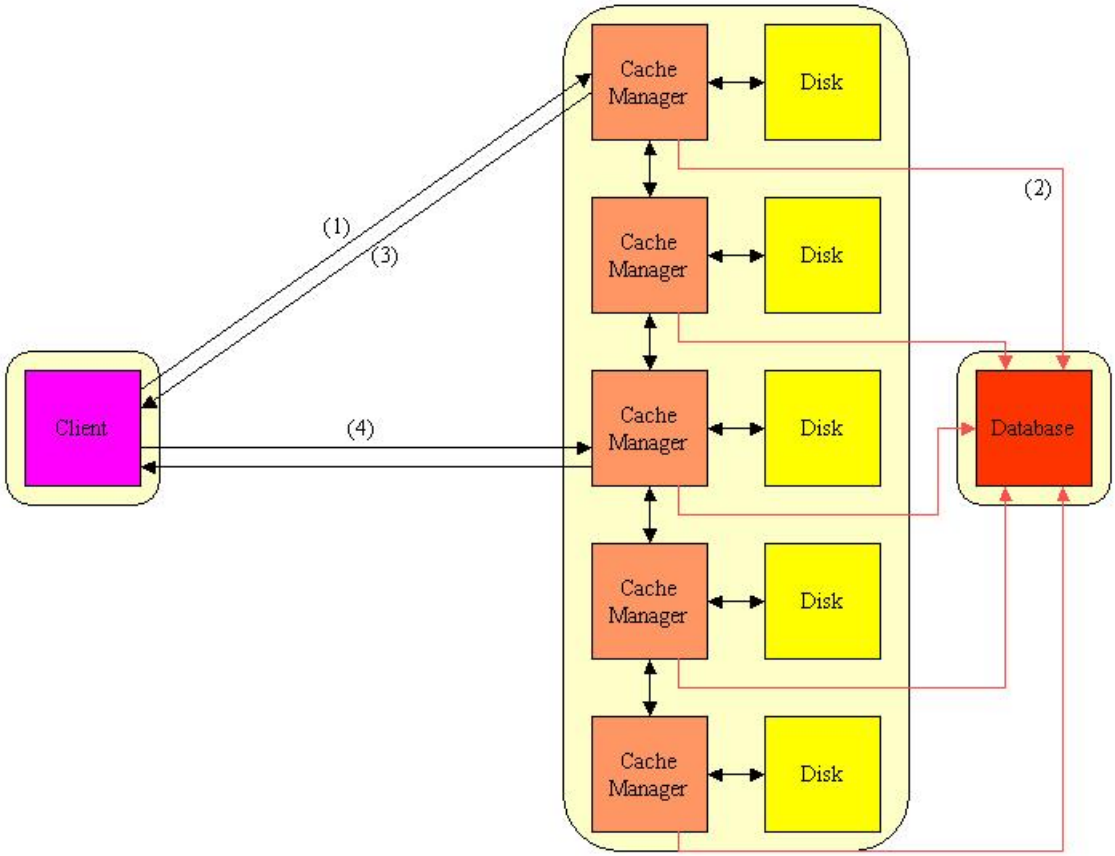

Figure 3.

The advantage to a protocol like this one is that it scales well. It is serverless in the sense any cache server can lookup a file's location, so there is no master file repository. Since the file transfer is direct between the cache server and the cache client, the introduction of additional cache servers to the network increases the overall throughput of the cache

system with no overhead. No single failure will bring the system down. Even the database may be replicated.

The terms "cache server" and "cache client" used above need some clarification. A cache client is a component that initiates the connection; a server is a component that accepts the connection. There are situations where one cache server is required to initiate a transfer on behalf of some agent. This third-party copy means a server must also occasionally act as a client, not an uncommon occurrence in peer-to-peer networking models. This kind of behavior requires transitivity of authorization credentials.

The cache management software is the generalization of three distinct uses for disk in the mass storage system. When we say disk, we typically mean a large pool of disks in some RAID arrangement.

A first type of disk storage is the user-invisible stage disk used with data movers as a buffer in front of tape drives. Files are added and removed from stage disks using a reference counting algorithm. A file is placed on a disk and its reference count is incremented for each thread using it. A file is only removed from the stage disk when its reference count is zero and it has the oldest access time of any file on the disk.

A second type of disk is the least-recently-used file cache. Files are added to this cache area (from which we get the overall cache disk name) by user request. These disks are perpetually full and contain a subset of the files stored on tape. These disks are divided into organizational groups that usually correspond to experiments. When a request is received by JASMine to add a new file to a group's cache disk space must first be made for the new file. This is done by determining the oldest file in the group by access time and removing it. This removal process is repeated until there is enough space for the file to be added.

This is a relatively heavyweight operation. The initiator finds the least recently used file by performing a JP scatter-gather operation in which each of the cache servers containing data for a group must independently search and find its least recently used file. This collection of files is then forwarded to the initiator as the gather portion of the operation. Since the multiple searches are done in parallel, the search time scales linearly with the number of servers. Such a distributed operation is also ripe with opportunities for race conditions, requiring careful attention to resource reservation.

Either of these first two types of disk could be used to pre-load data for farm jobs. We currently use a sufficiently large LRU disk pool to cache data from tape before we start batch processing of a job. This is advantageous because jobs are never blocked on our CPU farm waiting for tape. Our batch farm system enforces this staging making more efficient use of the CPUs.

The final use of the disk management software is for persistent data. There are certain classes of data that are added to a user-accessible area with no intention of removal. These typically represent the final output of physics data processing. These files, stored in the silo, are also cached online with no intention of removing them. For this we define

an explicit deletion model. Files are added to this disk group in the same was as to a cache disk, but no files are automatically removed.

3.11 Data Mover

The data mover is a collection of software components running on a single machine controlling at least one tape drive and running a job dispatcher. Data movers in a typical configuration are comprised of a dispatcher, a number of jobs, a cache manager for file staging, and a drive manager to coordinate access to local tape drives. Data movers are designed to be autonomous agents. They claim work to do from the queue.

The failure of a data mover reduces the overall bandwidth of the system by making some tape drives unavailable, but all other aspects of a failure are recoverable. Jobs that are marked as running on a failed mover will be reset to a known good state when the mover is brought online again. Partial file transfers between clients and data movers are detected and discarded by JP, the JASMine file moving protocol. A non-byzantine failure in a data mover (a kernel panic or failed power supply, for example) will result in a delay to any client that was currently being serviced, but not-yet started jobs will simply queue for other data movers.

With the emergence of SANs and other mechanisms for sharing devices that were historically dedicated to a single host we have taken into consideration the possibility that a tape drive could eventually have a presence on multiple hosts. We use our back-end database to coordinate drive allocation and locking. In the current implementation we use only host attached tape drives, so some complexity is avoided for now.

3.11.1 Cache Manager

The cache manager used within the data mover it the same as that described in section 3.10. The cache managers in the data mover, however, use the reference counting algorithm to free additional disk space as required for the staging of data. This algorithm prevents the accidental deletion of files staged to disk that have not yet been written to tape.

3.11.2 Drive Manager

There is one drive manager per data mover. The drive manager maintains database fields related to drives and issues drive objects to utilize the drives. It can be thought of as owning all drives that are local to the data mover. It performs the following services:

1. Mounts tapes in preparation for a job (contacts the Library Manager).
2. Dismounts tapes after idle timers expire (contacts the Library Manager).
3. Enforces drive related timers.
4. Set/release/enforce drive reservations.
5. Passes drive objects to jobs to allow jobs to perform tape operations.

3.11.2.1 Drive Reservation

Because the drive manager that owns the drive always handles drive reservations, the appropriate locking and avoidance of race conditions can be done using synchronization within the data mover's virtual machine. This is much simpler than disk space

reservation and job claiming, which must be done in a more distributed fashion using database table locking, database atomic inserts, or a distributed locking protocol, which is far more complex.

A drive can be reserved for reading only, writing only, use by a select volume sets, use by a select storage group, or some combination of these. To make sure that high priority users can write their data to tape immediately upon request, a drive can be reserved as write-only for the volume set that the users are using. This is simply done by maintaining reservation fields in the database for a given drive. Other components, however, query the drive manager for drive status. The dispatcher, for example, queries the drive manager for a list of available drives and their reservations. The dispatcher would then only start jobs that can make use of the drives available based on their reservations.

3.11.2.2 Drive

The Drive object is responsible for manipulating a loaded tape drive. It keeps track of file position on tape, seeks to a given location on tape, reads and writes data to tape to and from tape, and ejects a tape from a drive. When a job is passed a Drive object from the DriveManager, it simply uses OutputStreams and InputStreams from the Drive object to write and read data to and from tape.

3.11.3 Volume Manager

Volume managers provide the data mover and its components with information about volumes. This includes the type of volume it is, its location, its state, and its ownership. The volume manager is also responsible for allocating unused volumes to jobs when they need them.

3.11.4 Dispatcher

There is one dispatcher per data mover. It has the responsibility to inspect jobs in the Request database, inspect local resources, and then claim jobs as it can process them. Roughly, the dispatcher does the following:

1. Check that job starting is enabled.
2. Get the list of available drives from the Drive Manager
3. Check for secure, local disk space.
4. Get a list of pending requests.
5. Claim a drive (if failure, start over at (1)).
6. Reserve disk space (if failure, release the drive and return to (1)).
7. Claim the job(s) (if failure, release disk space, release drive, return to (1)).
8. Start the job thread.

By having a dispatcher on every data mover, the system will not stop dispatching jobs because of a single system crash or failure. This also cuts down on the amount of information on available resources a dispatcher must keep track of. The processing required to determine which job must run next is distributed across the data movers and thus lessens the CPU resources required by a single dispatcher.

3.11.5 Job

Jobs are user requests that execute in the context of a data mover, started by a dispatcher. The state of a job is recorded in the database in such a way that a job is always recoverable after a data mover failure. Jobs execute as an independent thread and are granted certain resources, perhaps a tape drive, a network bandwidth allocation, and portion of staging disk space. Error handling in jobs is important and the Java exception mechanism is used widely. Jobs do not necessarily have a one-to-one mapping with user requests. Sometimes, due to hardware differences on data movers, a request may be split into more than one job. The clients use multiple network connections to communicate with different data movers simultaneously.

3.12 Library Manager

The library manager is perhaps the simplest component of the system. It establishes an RMI service that mounts, dismounts, and queries the status of volumes in a library. JASMine's need for library-related information is minimal.

Our initial implementation of the Library Manager API was to a StorageTek Powderhorn library using ACSLS. Since there is no Java API for ACSLS we interfaced our code to it using JNI. This allowed us to create a multi-threaded abstraction of the underlying code.

3.13 Log Manager

With a system as distributed as JASMine, logging becomes a concern. System administrators need a central point for monitoring the overall system status, but this central point cannot be a point of failure. The Log manager classes provide the ability to record system activity to local log files and also over the net to a remote log server. The failure of the log server will make it unaware of log messages during its down time, but there is no other impact. Messages are logged at three different levels of severity: informational, error, and panic. Panic level messages are those that require system administrator intervention and generally send an automatic message to the on-call pager.

3.14 Configuration Manager

Another practical challenge to building a distributed system is configuration control. There are many runtime parameters that change infrequently but should not be compiled into the code. Initial prototypes that used configuration files turned out to be difficult to manage and required the instantiation of a configuration object in every virtual machine. This seemed unnecessarily burdensome.

Our solution to this problem was to create a configuration management class that used static methods to load configuration information from a database over the network and place them in the Java system properties. The first static lookup of a non-existent configuration parameter triggers the loading of the entire parameter set. Subsequent references use the cached information. This caching means that no persistent configuration server is needed. After the initial load, there is no failure possibility from the configuration manager.

In the database configuration parameters are arranged into global parameters and machine-specific parameters. The parameter load process selects the union of these

parameters that is correct for the calling machine. This allows global configuration changes to be made at a single point with no worry that some client was forgotten.

4 Future Work/Outlook

During the development of JASMine, some changes were suggested based on lessons learned from testing. In addition, other projects were started that influenced the design of JASMine.

The metadata stored in OSM needs to be imported into JASMine so that OSM can be decommissioned. Making use of newer tape drives and replacing OSM were the driving forces behind the development of JASMine.

Users have shown an interest in being able to access the data stored on Cache Managers via standard file copy protocols like ftp. To accommodate such requests, we will implement gateways to the Cache Managers that provide user interfaces for standard protocols to access data stored on them. Gateways will make implementing various protocols easier and will not require modifications to JASMine itself. These gateways will also provide external access to Cache Managers hidden behind firewalls. As part of Jefferson Lab's involvement in the Particle Physics Data Grid collaboration [5], we will provide interfaces to permit remote access to data on tape or on the local cache systems.

The naming scheme used to store data files within Cache Managers will be changed to include the entire full path name of the files original location unless the user provides an alternate name. This helps lessen the probability of two different files being given the same full path name for storage on disk. Although this is not a problem for JASMine alone, it might be a problem if the Cache Managers are used to store data for JASMine and other applications at the same time.

An XML based API will be created and implemented in JASMine. This will remove the requirement that clients be written in Java and use RMI or serialized Java objects for communication.

4.1 Web Clients

Users have shown great interest in having web-based clients that retrieve data files from JASMine. Such clients will authenticate the user and submit a request to get file or have files placed on a cache disk. In addition, web clients will be created to submit batch jobs to the batch farm. Such requests will gather all the required data files on cache disks before execution begins.

5 Conclusions

JASMine is a Mass Storage System that will scale to meet the needs of current and future high data rate experiments at Jefferson Lab. It is a lightweight, distributed, scalable, high performance, and high capacity system. The distributed and scalable design of JASMine make it suited to dealing with large volumes of data. The distributed nature of the Data Movers and Cache Managers are also essential to ensure the overall performance of the system. As the system is very modular, components can easily be used in other

environments; for example, the Cache Managers can be used as stand-alone disk pool managers or act as remote clients to JASMine.

References

[1] This research was sponsored by the U.S. Department of Energy contract DE-AC05-84ER40150. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policies.

[2] The Practice of Programming, Brian W. Kernighan and Rob Pike, Addison-Wesley professional computer series, Reading MA, 1999.

[3] The Java Native Interface: Programmer's Guide and Specification (Java Series), Sheng Liang, Addison-Wesley Pub Co; ISBN: 0201325772, 1999.

[4] American National Standards Institute, ANSI X.327-1987, "File structure and labeling of magnetic tapes for information interchange," New York, 1987.

[5] Particle Physics Data Grid Collaboration: http://www.ppdg.net