An Architecture for Lifecycle Management in Very Large File Systems

Akshat Verma*
IBM India Research Lab
akshatverma@in.ibm.com

Upendra Sharma
IBM India Research Lab
supendra@in.ibm.com

Jim Rubas
IBM Watson Research
rubas@us.ibm.com

David Pease
IBM Almaden Research
pease@almaden.ibm.com

Marc Kaplan
IBM Watson Research
makaplan@us.ibm.com

Rohit Jain
IBM India Research Lab
rohitjain@in.ibm.com

Murthy Devarakonda
IBM Watson Research
mdev@us.ibm.com

Mandis Beigi
IBM Watson Research
mandis@us.ibm.com

Abstract

We present a policy-based architecture STEPS for lifecycle management (LCM) in a mass scale distributed file system. The STEPS architecture is designed in the context of IBM's SAN File System (SFS) and leverages the parallelism and scalability offered by SFS, while providing a centralized point of control for policy-based management. The architecture uses novel concepts like Policy Cache and Rate-Controlled Migration for efficient and non-intrusive execution of the LCM functions, while ensuring that the architecture scales with very large number of files.

The architecture has been implemented and used for lifecycle management in a distributed deployment of SFS with heterogeneous data. We conduct experiments on the implementation to study the performance of the architecture. We observed that STEPS is highly scalable with increase in the number as well as the size of the file objects hosted by SFS. The performance study also demonstrated that most of the efficiency of policy execution is derived from Policy Cache. Further, a rate-control mechanism is necessary to ensure that users are isolated from LCM operations.

1. Introduction

Migrating data between different tiers of a multi-tiered storage system according to the changing quality of service (QoS) needs of data during its lifecycle is a well known problem[8]. Several commercially available products offer such hierarchical storage management [2, 4, 11]. However, increased governmental regulatory requirements on electronic data places new emphasis on the need for more complete policy control of data lifecycle management. Furthermore, the explosive growth of data in recent years requires lifecycle management solutions that are scalable to very large file systems with billions of files and that address not only the migration of data but also unchangeable retention and timely deletion.

The utility functions that comprise the lifecycle management solution must also be aware of their impact on the system since *24*x*7* service is now a common requirement. Therefore, there is a renewed interest in scalable, adaptive, and comprehensive lifecycle management solutions. In this paper, we present an architecture and implementation for lifecycle management (LCM) in a large-scale distributed file system that addresses these challenges.

1.1. Motivation for a comprehensive LCM Solution

The increased interest in scalable lifecycle management solutions stems from some new technologies that have emerged. The evolution of file level location independence supported in the new file systems like IBM SAN File System (SFS) [6] enables transparent movement of a file without affecting its users and (even running) applications. Another technological change comes in the form of Serial ATA drives that offer inexpensive, higher capacity, but lower performance storage alterna-

*Authors are listed in reverse-alphabetical order.

0-7695-2318-8/05 $20.00 © 2005 IEEE

IEEE Computer Society

tive to enterprise-class SCSI drives. Taking advantage of this new class of online storage, storage administrators deliver differentiated QoS by migrating data between the storage classes as the performance requirements of data changes over time. To take the simple example of an email server, the recent messages may be hosted on SCSI drives when they arrive and moved to cheaper SATA drives as they become older (and their access probability goes down). These technology trends have led to an environment where the lifecycle of a file may involve many migrations. This is a stark contrast to the typical file lifecycle model for which existing lifecycle management solutions are designed; i.e., creation followed by backup and archival, a model where the file data would move only once, usually from a SCSI device to a tape pool. These additional complexities in file lifecycle require us to develop efficient infrastructure for large-scale lifecycle management that can deal with a large number of migrations.

Also, in existing lifecycle management solutions, there often are multiple tools designed to control the various aspects of the file lifecycle. For example, backup and archival to tape, migration to near-line storage, and migration to WORM storage are under the control of different tools. Managing multiple tools to provide an integrated lifecycle strategy is complex, error-prone and can overburden the job of an administrator. Further, current lifecycle management solutions are not integrated within the file system but instead exist outside the file system thereby limiting their ability to take advantage of file system internal structures for scalability. In large scale file systems with billions of files, selecting candidate files for lifecycle management operations can be extremely resource intensive. Naive implementations scan the entire file system namespace for candidate files on every policy invocation. Studies [3] show that less than 1% of files are modified every day in a file system with 200,000 files, and the percentage reduces with increasing size of the file system. Our measurements show that roughly 5000 files can be scanned per second, which implies that a naive implementation requires 27 hours to scan one billion files on every policy invocation.

Hence, one needs to design file lifecycle management solutions that are scalable, provide an integrated control for all management functions, and has mechanisms to ensure that the large number of migrations do not disrupt regular client traffic.

1.2. Contribution

We present the Storage Tank Enhanced Policy Service (*STEPS*) architecture for policy-based file lifecycle management that provides an integrated and central management control for all lifecycle management functions in a non-disruptive manner. In order to capture the diverse lifecycle management functions, we have provided a powerful yet simple policy language that storage administrators can use to specify policies to control all aspects of file lifecycle management. The architecture provides centralized policy-based management for controlling the placement of files in different storage tiers, from their initial creation to their eventual deletion or secure archival. The policy execution is designed to scale to billions of files while minimizing the impact on ongoing client workloads.

To avoid the scanning of the entire filesystem metadata for each policy execution, we propose a novel concept of a Policy Cache in which the entire metadata is scanned at policy initialization time and a cache is built of future policy actions applicable to each file. The cache is updated subsequently, using a lazy and batched approach, as a result of modifications to files, file creations and deletions. This significantly reduces the cost of policy execution as the problem of candidate file selection is now reduced to cache lookup. In our example of a file system with one billion files, our policy cache will reduce the scanning to the actual set of files modified, which is likely to be less than 1%; thus it will take less than 15 minutes to find candidates for LCM actions even in this huge file system.

We also provide a resource arbitration mechanism for controlling the rate of LCM initiated data movement in order to minimize the impact of LCM operations on normal client operations. LCM operations often require migration of large amounts of data between storage tiers. For very large scale file systems, the time required to complete data migration may be several hours during which client access to the system could be severely affected. For those large *24*x*7* service providers, such an impact would have a negative affect on the business. We provide a control-theoretic mechanism to adapt the rate of migration according to changes in client workloads so as to minimize the impact on them.

We have implemented this architecture in The IBM SAN File System (SFS) also known as Storage Tank [6]. SFS is a distributed file system for SAN-attached storage that supports heterogeneous clients. Our implementation leverages advanced management concepts provided in SFS such as separate centralized metadata management, namespace partitioning into containers, and storage pools. However, the architecture is general enough to be applicable for implementation in other distributed file systems. Experimental results validate that the *STEPS* architecture is scalable and the rate-control mechanism is effective in ensuring that LCM activities do not disrupt regular client traffic.

IEEE COMPUTER SOCIETY

The rest of this paper is organized as follows. In section 2, we present a detailed design of our system along with a discussion on the design choice that we made. In section 3, we present the details of our prototype implementation, our performance study and results regarding the scalability and efficiency of the implementation. Finally, we conclude with our key observations in section 4.

2. Framework and Architecture for SAN File System Policy Service

We describe a policy-based lifecycle management architecture, Storage Tank Enhanced Policy Service (STEPS), in the context of the IBM SAN File System [6], a distributed filesystem that can support a large number of file objects. We first provide a brief overview of the IBM SAN File System (SFS).

2.1. SFS Overview

The IBM SAN File System is architected to be a highly available file system for SAN-attached storage; it is designed to provide a network-based, heterogeneous file system for data sharing and centralized policy-based storage management in an open systems environment. SFS is designed to enable host systems to plug into a common SAN-wide file structure (Fig. 1). With SFS, files and file systems are no longer managed by individual computers; instead, they are viewed and managed as a centralized IT resource with a single point of administrative control. SFS provides a common file system for UNIX, Windows and Linux servers, with a single global namespace to help provide data sharing across servers.

SFS maintains all the file system metadata on a dedicated metadata server (MDS) cluster. SFS clients retrieve the physical location of a particular file segment from the MDS, and directly fetch the data from the disks attached to the SAN. SFS also uses dedicated SFS clients to perform any bulk data copy for LCM functions, thus ensuring that data access operations never flow through the metadata servers. The server-free data path along with the aggressive caching of metadata supported by SFS ensures that data access is not affected greatly by server congestion, thus making the system highly scalable without the limitations normally associated with Network File System (NFS) or Common Internet File System (CIFS) implementations.

Unlike many file systems, the name space in SFS is completely decoupled from the storage space; that is, a file's location in the name tree has no connection with its location in the storage subsystem. The name space is subdivided into segments called "containers"; containers are used for various management purposes in SFS, including server load assignment. The storage space is subdivided into "pools"; pools are named collections of storage volumes; at any point in time, a file's data is stored in one pool.

The metadata servers designate one of their number as a master node with all the other servers being subordinate to it. At a file system level, the master MDS assigns containers to individual subordinate nodes, which are responsible for managing the assigned container. The subordinate node that manages a specific container allocates space for the files belonging to that container; a file's data is allocated on one or more of the volumes of the appropriate pool during its lifetime, as specified by the LCM policies. The distributed server cluster and random master election provide a high level of concurrency, that in turn ensures scalability of SFS with large number of files, and fault tolerance in the presence of server failures.

However, the distributed nature of the MDS cluster presents challenges in designing an LCM policy infrastructure that can be controlled centrally without affecting the scalability and the high degree of concurrency in the SFS. Moreover, the policy infrastructure should not violate the intrinsic design principle that management components do not sit in the datapath. Further, LCM operations like backup and archival that deal with tape or optical pools require us to integrate the policy architecture efficiently with a (possibly) third party tertiary storage manager. We now present an architecture that achieves the above goals.

2.2. The STEPS Architecture

The *STEPS* architecture has a policy user interface that is used by an administrator to create "high level" lifecycle management policies. A policy is a tuple of a condition and an LCM action (migration, replication, deletion, backup or archival) on a datagroup. A datagroup is a collection of files that share certain common properties; it is specified by a boolean expression of file attributes. An example of a common policy used for LCM is

If <space utilization of PREMIUM_POOL is greater than 80% > *then <MIGRATE *.tmp files greater than* 1 *MB ordered by size to IDE_POOL until the space utilization of PREMIUM_POOL is* 60% *".*>

The datagroup is defined as the set of files whose name matches the pattern *.tmp and whose size is greater than 1 MB, ordered by size.

The central control in our architecture (Fig. 2) lies within a File Policy Scheduler and Orchestrator (*FPSO*) that determines if a particular policy is active at any point

0-7695-2318-8/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

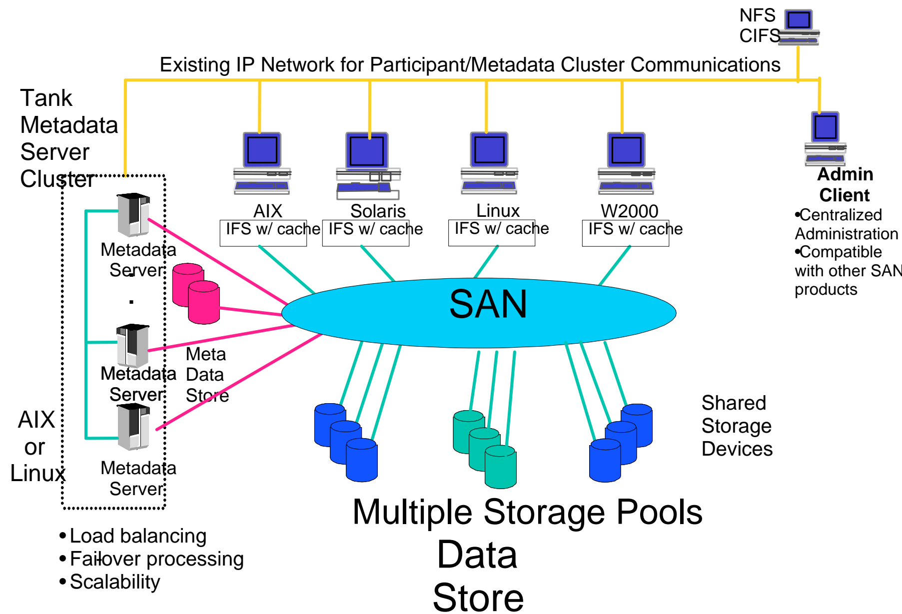

Figure 1. A typical SFS Deployment

in time. *FPSO* uses File Policy Agents (*FPA*) to execute the policies currently active. A *Scanner* component on each server is responsible for initializing and updating the Policy Cache. The *FPA* queries the *Scanner* to obtain the list of files belonging to the datagroup on which the particular policy action is defined and uses a File Policy Effector (*FPE*) to perform the required LCM function.

The control flow for a given policy execution consists of the following steps: The "high level" policies are transformed into "low level" policies particular to *FPSO* and *Scanner* components. In the above example, the *FPSO* policy is *If the space utilization of PREMIUM_POOL is greater than* 80% and the *Scanner* policy is the definition of the datagroup, **.tmp files greater than* 1 *MB ordered by size*. The *Scanner* low level policies are additionally transformed into SQL predicates in order to use the SQL engine provided by SFS. The *FPSO* monitors the pool performance and space parameters along with temporal data and sends periodic requests to the *Policy Decision Manager* for policy evaluation. In response, the *Policy Decision Manager* returns the applicable policy action to be executed. The *FPSO*, after some initial bookkeeping, forwards the action to the *Agents* where the action is applied to their respective containers. *Agents* obtain the list of candidate files eligible for policy execution from their local *Scanner*. *Agents* communicate among themselves to further prune the file list and obtain the exact set of files on which the LCM action corresponding to the policy should be applied.

The *FPE* uses the SFS clients for those LCM operations that involve only the disks (migration, replication and deletion) and invokes a tape manager (possibly third party) for other LCM operations (backup and archival).

2.3. Centralized FPSO and Distributed Agent

A key design choice in the *STEPS* architecture is to separate out the multi-server management functions from the per-server management functions. Towards this purpose, we have a File Policy Scheduler and Orchestrator (FPSO) component that co-ordinates the overall execution of a complete policy across all MDS nodes. The actual execution of a policy is managed by File Policy Agents, with one agent deployed on each MDS effected by the policy.

The *FPSO* component performs the following functions:

- Seeks policy guidance from the Policy Decision Manager based on time and SFS thresholds specified in the policy (e.g., space threshold, throughput, response time).
- Passes any active policy to File Policy Agents on metadata servers for execution/enforcement and ensures fault tolerance from agent failures.
- Provides a centralized point to monitor all active LCM actions.

Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005)
0-7695-2318-8/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

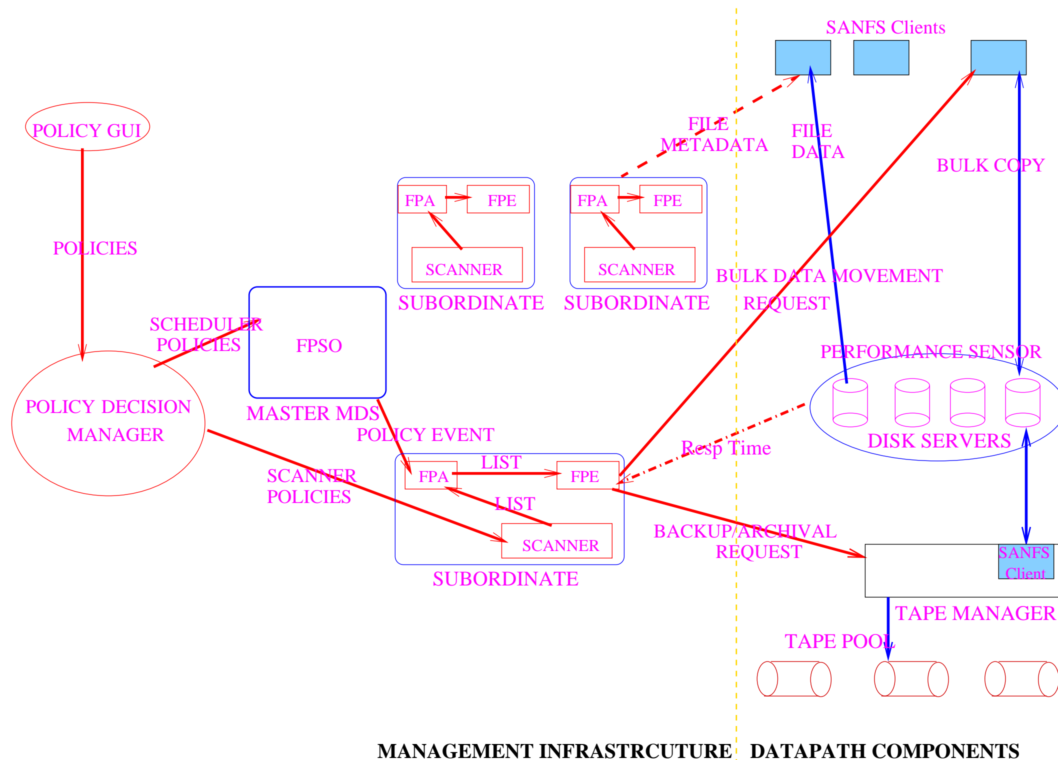

Figure 2. The STEPS Architecture

- Decide the SFS clients to be used for bulk data movement related to any specific policy action.

Our architecture supports a centralized *FPSO* (on the master MDS) since the policies apply uniformly to all the containers and hence to all the metadata servers that manage the containers. With this design, we do not need to monitor if a particular policy has become active at multiple servers and therefore a single *FPSO* suffices. A single *FPSO* also ensures that management functions offer a centralized control point. In the distributed SFS architecture, a single policy may affect containers that are distributed amongst many servers in the cluster. Therefore, for each LCM policy, we spawn an *FPA* agent at each of the subordinate nodes that are affected by the policy.

A distributed agent architecture requires us to handle the additional complexity of inter-agent communication, which is necessary for policies that require a global ranking of files that need to be moved. To elaborate, in the example policy specified earlier, the agents need to compute a set of *.tmp files that is large enough to bring down the space utilization on *PREMIUM_POOL* from 80% to 60% while ensuring that all *.tmp files that are moved from *PREMIUM_POOL* are larger than the tmp files that have not been moved. Hence, the *Agents* need to compute a global ranking of the files in the distributed architecture.

However, having a distributed architecture allows us the high degree of concurrency supported by SFS by using all the servers in a cluster for any policy action. Moreover, by delegating the LCM functionality for each container to the node that manages its metadata, we solve the problem of balancing the *LCM* load across the servers without any additional effort. The parallelism offered by the distributed architecture allows the system to scale well to a large number of files by dividing the execution across multiple servers. A single computationally expensive policy does not slow down access to a metadata server or to the containers managed by that particular metadata server.

2.4. File Policy Cache for Efficient Policy Execution

The most computationally expensive operation in the *STEPS* architecture is scanning the metadata to determine the files that belong to a particular datagroup. For a server that manages a very large amount of metadata (e.g. hundreds of millions of files), a naive implementation would burden the server – both the policy operations as well as client accesses to the server would be affected significantly. Hence, the *STEPS* implementation main-

tains a *Policy Cache*. The *Policy Cache* contains, for each file, the policies that apply to the file, and the times at which each policy calls for an action (eg. migration or deletion.)

The Policy Cache can be viewed as a database table where each row is a triple: rule number, predicted time, file object id. (The file object id in SAN FS is a unique identifier similar to an inode in most Unix file systems). A Policy Cache entry with values (R, T, I) means that file I should be selected as a candidate for application of rule R at time T. The Policy Cache is indexed two ways: (1) by rule number and predicted time and (2) by file object id. The first index allows one to quickly determine which files are subject to a given rule within a given time interval. The second index allows one to quickly locate all entries for a given file. To be able to maintain a Policy Cache *a priori*, the Scanner requires the definition of the datagroups. These are passed to it as Scanner Policies by the Policy Decision Manager, at the time of creating of the policies.

In order to populate the *Policy Cache* initially, the Scanner checks all the files in any container assigned to its corresponding MDS against the policy rules set, in order to determine which rules apply to the file at what time. Although this process can be lengthy it need only be executed once during Policy Cache initialization (but again should some rules change.) Naturally, changes to files "invalidate" the corresponding entries in the *Policy Cache*. So the *Policy Cache* is brought up-to-date periodically and/or on-demand by re-evaluating the policy rules, but only for the files that have changed.

The *STEPS* Scanner determines which files have changed by examining a changed files index that is maintained by SFS. The SFS changed files index is comprised of hourly "buckets" - the first time a file changes within a given hour SFS logically moves the file's index entry to the bucket representing all files that have changed during the current hour of the current date.

Hence, the Scanner overhead to update the *Policy Cache* is proportional to the number of changed files per hour, **not** the total number of files, nor is there a penalty for rapidly changing files. Various techniques keep the cost of maintaining the changed file index very low;the details of which are beyond the scope of this paper.

2.5. Rate-Controlled Bulk Data Movement

The information life cycle management functions (migration, replication) require moving large amounts of data at regular intervals. This places huge demands on the storage resources, mainly disk and controller bandwidth, and may affect client performance to such an extent that their QoS requirements are not met. In order to ensure that the I/O performance as perceived by the client is not affected, we control the rate at which such bulk data is moved.

The *STEPS* architecture uses sensors to monitor the performance parameters, namely response time and throughput, for each volume. The *FPSO* computes a target response time for each storage pool such that the performance requirements of all the datagroups hosted on that pool are met. The target response time thus computed for each pool is also the target response time for each volume in the pool. The total throughput offered by the volume is then controlled by the Effector (*FPE*) in order to ensure that the target response time is met. If the current response time of the volume is greater than the target response time, migration is throttled back whereas if the response time of the volume is less than the target, migration throughput is allowed to increase.

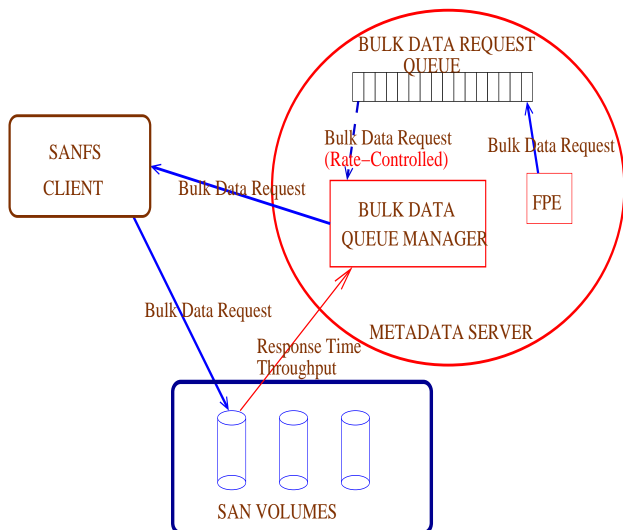

Figure 3. Rate Control Mechanism

The rate control is enforced inside the SFS metadata server (Fig. 3), which maintains a separate request queue for bulk data movement requests. A particular migration throughput is ensured by interspersing periods of bulk data movement with periods of sleep in the queue manager of the bulk data movement queue. We use a linear rate controller to vary the migration throughput in order to ensure the response time of the volume meeting its target. The basic feedback controller is described in Eqn. 1

$$R_i(t) = \frac{R_i(t-1)\delta_i^T}{E(\delta_i^t)} \quad (1)$$

where $R_i(t)$ is the migration throughput at time t, δ_i^t is the response time in the interval $(t-1, t)$ and $E(\delta_i^T)$ is the target response time for volume V_i. For a bulk data

IEEE COMPUTER SOCIETY

movement from Volume V_i to V_j, the rate of migration $R_{i,j}(t)$ is then given by $\min\{R_i(t), R_j(t)\}$.

We have observed in our preliminary investigation that response time is not a stable parameter and may shoot up for small periods of time, independent of migration throughput. In order to ensure that migration is not throttled due to transient increases, we also incorporate throughput served by the volume as another parameter for stability. Hence, migration is rate-controlled only when the total throughput to the volume is more than a threshold, where the chosen threshold is computed so as to ensure that the average response time for 99% of the intervals or more is less than the target response time. Incorporating throughput in the rate-control framework has the added benefit of providing damping and reducing oscillations in rate of migration. Our architecture is designed to support the sophisticated *QoSMig* [1] methodology that provides optimal client performance while ensuring that LCM objectives, such as meeting a particular deadline, are satisfied.

3. Implementation and Performance Evaluation

We now describe our implementation of the *STEPS* architecture and an experimental study that demonstrates its scalability and efficiency.

3.1. STEPS Implementation

We have a distributed deployment of IBM SAN File System and have implemented the *STEPS* architecture on top of it. Our SFS test bed consists of two metadata servers hosted on Intel Xeon 2.8 GHz servers with 1GB of RAM and 20 GB of attached disk storage. We use an Intel Pentium 4 2.66 GHz machine with 1 GB RAM as an SFS client dedicated to file movement pertaining to LCM activity. We currently host two pools for providing differentiated QoS. The logical volumes comprising the *PREMIUM_POOL* pool are hosted on Seagate 36GB enterprise class SCSI disks. The *IDE_POOL* consists of Maxtor 40GB IDE disks that host data with lower performance requirement. Both sets of disks are controlled by a FAStT200 [9] storage server. Our implementation has mechanisms for recovering from node failures and disk failures by maintaining the LCM operations in progress on stable storage.

The performance metrics such as response time of a volume needed for rate-control mechanism are not available from the FAStT200 storage server. Hence, we instrument the QLogic host bus adapter (HBA) driver on each SFS client to measure the response time and throughput seen by the client for each volume that is accessible to the client.

3.2. Experimental Setup

The experiments were conducted by hosting a file system with two containers on the *STEPS* implementation. The first container mimics a workplace setting with production code consisting of C/C++ source files and object files. The other container consists of multimedia (audio and video) files that are accessed by an mpeg player (for Linux) [10]. The mpeg player can tolerate average response time latencies of 20*ms* and that is defined as the target response time for client access to this container for rate-controlled migration. The production file system has policies for moving older files from the *PREMIUM_POOL* pool to the *IDE_POOL* pool. Moreover, policies are deployed to ensure that files on *IDE_POOL* pool that are accessed are moved back to the *PREMIUM_POOL* pool.

We study the performance of *STEPS* architecture in the following contexts: (i) scalability with increase in file objects, (ii) efficiency of the *Policy Cache*, and (iii) usefulness of the rate-controlled migration. For the first set of experiments, we increase the number of file objects managed by a metadata server from 100 to 100,000. In the second set of experiments, the efficiency of policy cache is established by repeating the first set of experiments; this time with the policy cache disabled. The difference in the two sets of numbers describe the performance improvement that can be directly attributed to the *Policy Cache*. In a third set of experiments, we observe the performance of the mpeg player with the rate-controlled mechanism activated and contrast it with the response times seen by the mpeg player when the rate-control is deactivated.

For all these experiments, the number of LCM Candidate files was kept at 10% of the total number of files. The measurements are averaged over 10 runs, and the policy execution time excludes the actual data movement time.

3.3. Results

The *STEPS* architecture scales well as the number of file objects increases, with the policy execution time increasing from 55 ms to only 150 ms (Table 1) as the number of file objects managed by a metadata server is increased from 100 to 100,000. We note that the scalability of the architecture is a direct consequence of the Policy Cache. We observe that when the Policy Cache is disabled, policy execution increases almost linearly with increase in the number of file objects taking as much

0-7695-2318-8/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

Table 1. Policy Execution Time with increase in file objects when Policy Cache is enabled and disabled

Number of File Objects	Policy Cache Enabled	Policy Cache Disabled
100	55 ms	100ms
1000	62 ms	1sec
10000	70 ms	3 sec
50000	100 ms	11 sec
100000	150 ms	21 sec

as 21 seconds when the number of file objects reaches $100,000$. This is because the complete metadata has to be scanned during execution of the policy if the policy cache is disabled. On the other hand, with the policy cache enabled, the execution of a policy only requires looking up a cache of eligible files with no computation for candidate selection.

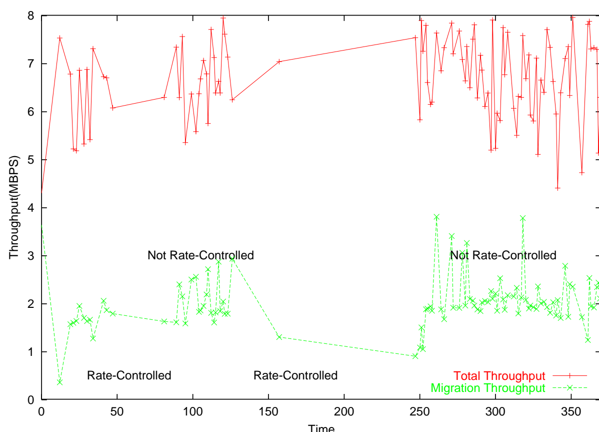

Figure 4. Throughput (Total throughput and migration throughput) with and without Rate-control

Finally, we investigate the efficacy of the rate control mechanism for migration. Fig. 4 and Fig. 5 present a snapshot of the system as the rate-control mechanism for migration is activated and deactivated. Fig. 4 shows the total throughput as well as the migration throughput and Fig. 5 shows the storage subsystem response time for the same period. We observed that while our rate-control mechanism is active, the response time (averaged over 1 second) for the mpeg player remained below $20ms$ and hence the mpeg player played the video without any glitches. In the absence of the rate-control mechanism, the average latency for the player was consistently above $100ms$ due to resource contention from the LCM work-

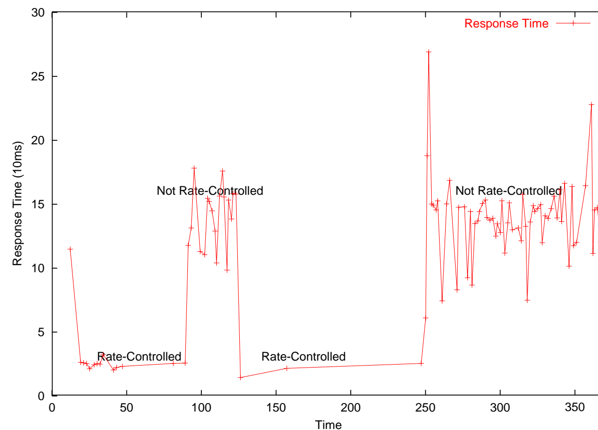

Figure 5. Response Time of the mpeg stream with and without Rate-control

load, and as a result the mpeg player dropped frames and the video was jittery.

We also observe that the response time requirements are met without completely throttling the LCM bulk data movement. In fact, at times, with only a slight decrease in migration bandwidth, we were able to meet the target response time. Hence, the rate-control mechanism is critical for ensuring that client performance does not suffer significantly due to the additional load generated by the LCM operations.

4. Conclusion and Future Work

We presented a framework and *STEPS* architecture for automation of lifecycle management functions in large distributed filesystems using administrator defined policies. We have shown how to leverage the distributed architecture of a scalable file system, such as IBM's SAN File System (*SFS*) to design an LCM architecture that scales well with large number of file objects, without imposing a significant load on normal file operations.

We also presented a prototype implementation of the *STEPS* architecture in the context of *SFS*, and observed experimentally that the architecture is highly scalable and efficient. Our experiments pinpoint the specific features in the architecture that are required to ensure that the system scales well and client workloads are isolated from LCM workloads. We believe that our framework will fuel further research directed towards making large storage systems self-managing in nature.

An interesting direction of research that we are investigating is to *a priori* predict the load that LCM operations would put on the storage subsystem at any time in the future. This would allow the system administrator to identify times of possible overload in the system and

IEEE COMPUTER SOCIETY

take appropriate actions (by rescheduling some of them) to avoid overload.

We note that the *Policy Cache* provides us the ability to predict the file set that is affected by any LCM policy. Hence, we have the information of the load generated by any policy; provided no file activity takes place in the interim. We plan to use short-term prediction tools to establish trends in file activity and use them along with the current file set, as predicted by the *Policy Cache*, to predict the candidate file set for the LCM policy at the future time.

5. Acknowledgment

We would like to thank Linda Duyanovich, Aki Fleshler, Sugata Ghosal, Jim Seeger, Ronnie Sarkar and Jason Young (IBM) for their valuable suggestions.

References

[1] K. Dasgupta, S. Ghosal, R. Jain, U. Sharma, and A. Verma. (2005). QoSMig: Adaptive Rate-Controlled Migration of Bulk Data in Storage Systems. To be published in *Proc. International Conference on Data Engineering*, 2005.

[2] J. P. Gelb. System-Managed Storage. In *IBM Systems Journal*,28(1),1989.

[3] T. J. Gibson, E. L. Miller, and D. D. E. Long. Long-Term File Activity and Inter-Reference Patterns. Computer Measurement Group (CMG 98) Proceedings, 1998.

[4] M. Kaczmarski, T. Jiang, D. Pease. Beyond Backup Towards Storage Management. In *IBM Systems Journal*, 42(2):322-338, 2003.

[5] D. H. Lawrie, J. M. Randal, R. R. Barton. Experiments with Automatic File Migration. In *IEEE Computer*, 15(7): 45-55, 1982.

[6] J. Menon, D. A. Pease, R. Rees, L. Duyanovich and B. Hillsberg. IBM Storage Tank- A Heterogeneous Scalable SAN File System. In *IBM Systems Journal*, 42(2):250-267, 2003.

[7] E. L. G. Saukas, and S. W. Song. Efficient Selection Algorithms on Distributed Memory Computers. In *SuperComputing*, 1998.

[8] A. J. Smith. Long Term File Migration: development and evaluation of algorithms. In *Commun. ACM Vol.24, No.8 :521-532*, 1981.

[9] Anonymous. IBM TotalStorage Products. http://www.storage.ibm.com.

[10] Anonymous. Mpeg Player for Linux. http://mplayer.com

[11] Anonymous. Veritas Data Protection Products. http://veritas.com. 2004.

0-7695-2318-8/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY