Adaptive Resource Management for Scalable Network–Attached Storage Systems

Konstantinos Kalpakis[1]**, Koustuv Dasgupta, and Shamit Patel**
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, MD 21250, USA
{kalpakis,dasgupta,spatel22}@csee.umbc.edu

Keywords: resource management, replication, hashing, load balancing, Network–Attached Storage

ABSTRACT

The growth in the commercial use of the Internet and the proliferation of data-intensive network services, have heightened the demand for large–scale storage systems. Over the last few years, Network–Attached Storage has emerged as a basic tenet for simplified storage management and improved scalability, reliability and performance of storage systems.

We propose a novel technique for efficient resource management in such systems, with the objective of minimizing response time while maximizing throughput. The two salient features of our approach are: (a) intelligent replication of data objects across multiple disks based on client access patterns, and (b) load balancing among the disks using hashing. Our approach is easy to implement in a decentralized manner. Further, we present simulation results of the proposed scheme using synthetic workloads. Our approach is shown to significantly outperform various alternatives with respect to response time and throughput as the load on the system increases, while creating only few replicas. The resulting system is scalable and self–configurable, and adapts to changes in the operating environment quickly.

INTRODUCTION

The rapid growth of the World Wide Web and the proliferation of data-intensive network services have led to today's digital *tsunami*. Data sets stored online are growing at a phenomenal rate, often reaching several terabytes and doubling every year [3]. Popular examples include Internet data centers, repositories of satellite and medical images, e-commerce companies and multimedia entertainment content. Unfortunately, such applications impose stringent requirements that are difficult to meet using current storage systems. They require high availability, performance that enables servicing millions of clients with low latencies, scalability that matches the growth of the user base and the data they need to access, and minimal maintenance and administration costs. Consequently, a number of recent research efforts have been directed towards the investigation of techniques for building scalable storage systems.

Commercial systems interconnect storage devices and servers with dedicated Storage Area Networks (like Fibre Channel), and enable incremental scaling of bandwidth and capacity by adding more storage to the network. Advances in LAN performance have created an opportunity to take a similar approach using a general-purpose LAN as the storage backbone. Such systems are built from disks that are distributed throughout the network and attached to dedicated servers, cooperating peers, or the network itself. As pointed out in [4], they can be classified into two broad groups. One group of scalable storage systems, e.g. Frangipani/Petal [11], layers the file system functions above a self–manageable network storage volume using a *shared disk model*. Policies for data striping, redundancy, and storage site selection are done on a volume basis; cluster nodes coordinate their access to the shared storage blocks using a locking protocol. The second group, widely known as Network–Attached Storage system, is composed of intelligent 'plug and play' [6] disks that connect directly to the LAN. Specifically, the CMU system [9] adopts the separation of file managers, e.g. the name service, from block storage services to enable direct high–speed data transfer between end clients and Network–Attached Disks (NADs or simply disks). In related work, Seagate [2] argues for the value of and requirements for Object Oriented Devices in support of Network–Attached Storage.

The importance of large scale storage system performance has further led to a plethora of manual and ad-hoc techniques to ensure high availability and balanced load. In

[1]Supported in part by NSF under grant IRI–9729495, and by NASA under Cooperative Agreement NCC5–315 and Contract NAS5-32337.

demanding environments, adding more storage to the network simply seems to be a temporary solution. An ideal storage system should be driven by automated techniques that aim to meet demand surges by *efficient resource management*, in terms of storage and processing capacity of the disks. Specifically, we claim that the performance of Network–Attached Storage systems depends critically on (a) intelligent replication of data objects across multiple disks, and (b) assignment of data objects to disks in order to balance the load across them (or optimize a more complex objective function). Moreover, any such assignment is likely to change over time when either the workloads, i.e. client access patterns change, or when new disks are added/removed from the system, or when existing disks fail/recover.

In this paper, we first present novel algorithms to calculate the number of replicas for each data object, so as to minimize the average load on the existing disks. Replication decisions are based on the popularities of individual objects, and designed to satisfy the total demand for all the objects within the storage capacity constraint of the system. Second, we employ hashing to place the replicas at different disks and distribute the requests among the replicas, so as to achieve judicious load balancing among the disks. Our algorithms are lightweight, distributed, and dynamic in nature. We further demonstrate how the proposed scheme can adapt to changes in the operating environment (like addition or removal of disks) with minimal overhead.

The rest of the paper is organized as follows. We first give a brief overview of the system model and provide a formulation of the problem as a constrained optimization problem, along with detailed descriptions of our proposed algorithms. Next, we present a comparative analysis of the performance of the algorithms in a simulated Network–Attached Storage system. Finally, we conclude the paper.

SYSTEM DESIGN

We model the Network–Attached Storage system as having four fundamental components, namely (a) the network, (b) the client, (c) the network-attached disks , and (d) an object–oriented file system [2]. The network is a high–speed any-to-any interconnect for client to disk or disk to disk. Each disk has a processing capability which can be exploited for uses other than data transfer [9]. Disks store the data in form of objects and associated attributes. An object–oriented file system allows operations on objects and their attributes. The client has a unified view of the entire file system and issues requests to the disks. We assume that the following operations can occur in the system : create, delete, get attributes, set attributes, read and write of an object, as well as create and delete a replica of an object. For read(get)/write(set) operations on objects we assume that the read one–write all (operational) replicas strategy is used. We next show how the system calculates the number of replicas, decides on their placement, distributes the requests among replicas, and adapts to changes in the number of disks.

Replica Calculation

We are given n objects $x_1, x_2, \ldots, x_n$. Each object x_i has a size s_i. Let the number of requests for x_i be given by λ_i, and let k_i be the total number of copies (replicas) for x_i. Let $S = \sum_{i=1}^{n} s_i$ be the total size of all the objects. Suppose that we are given m disks $D_0, D_1, \ldots, D_{m-1}$, with each NAD having total size Δ and having R space available for replication. Furthermore, we associate a weight w_i with each request for object x_i. The weights of the requests can be chosen in a number of ways. We consider three cases of weights for the requests: (a) the *uniformly–weighted* case, where $w_i = 1$ for each object x_i, (b) the *size–weighted* case, where $w_i = s_i$ for each object x_i, and (c) the *work–weighted* case, where $w_i = 1 + \alpha s_i$ for each object x_i, and α is some constant. The constant α can be chosen to be equal to the ratio of the time to transfer 1 unit (eg. KB) from/to a disk over the seek time.

We want to find the number of replicas for each object so that the weighted average of the number of requests each replica services is minimized. With an appropriate placement of those replicas among the various disks, the weighted average of the number of requests each disk services will also be minimized. We can see that, (a) in the uniformly–weighted case, the average number of requests serviced by a disk is minimized, (b) in the size–weighted case, the average amount of data transfered by a disk is minimized, and (c) in work–weighted case, the average amount of work done by a disk is minimized. Note that (a) is appropriate when the seek time dominates the transfer time for all the objects, (b) is appropriate when the seek time is negligible with respect to the transfer times, and (c) is appropriate in the other cases.

Next, we describe algorithms for minimizing the average weighted number of requests a disk services, which is given by

$$\frac{1}{m} \sum_{i=1}^{n} \frac{\lambda_i w_i}{k_i}. \tag{1}$$

The total number of replicas k_i each object x_i has should be at least one, since otherwise an object is lost from the system, and should be no more than the number of disks, since it is not beneficial for any single disk to have more than one copy of any particular object. Thus, the total number of replicas for each object x_i should satisfy the constraint

$$1 \leq k_i \leq m. \tag{2}$$

Moreover, the space needed to store all the replicas of all the objects should be no more than the total space necessary for a single copy of all the objects plus the total space available for replication among all the disks

$$\sum_{i=1}^{n} k_i s_i \leq S + mR, \tag{3}$$

where we assume, without loss of generality, that $S + mR \leq m\Delta$. Clearly, the problem then is to find integers k_i, $i = 1, 2, \ldots, n$, that minimize (1) subject to the constraints (2) and (3). We call this problem the *Minimum Disk Load Replication* (MDLR) problem. It can be shown that the MDLR problem is NP–complete (by showing that the Knapsack problem, which is known to be NP–complete, is a restriction of the MDLR problem; details omitted due to space constraints).

Since the MDLR problem is NP–complete, we are interested in finding approximate solutions close to the optimal of the MDLR. To this end, we consider the continuous relaxation of the MDLR problem, i.e. when the k_i can take continuous (real) values; we call this problem the c–MDLR problem. Given an optimal or approximate solution to the c–MDLR problem, we can construct a good approximate solution to the MDLR problem.

The c–MDLR problem is a convex optimization problem, since it has a convex objective function and concave constraints. It is known that all local optima of a convex optimization problem are also global optima [8]. An (optimal) solution for the c–MDLR problem can be obtained using the *active set* method [8]. Finding the global minimum of c–MDLR using the active set method is computationally expensive, and it is not easily done in a decentralized manner.

Thus, we consider the relaxation of the c–MDLR problem obtained by removing the constraint (2) (and, without loss of generality, tightening (3)), i.e. minimize

$$\frac{1}{m} \sum_{i=1}^{n} \frac{\lambda_i w_i}{\hat{k}_i}, \tag{4}$$

subject to the constraints $\hat{k}_i \geq 0$ and

$$\sum_{i=1}^{n} \hat{k}_i s_i = S + mR. \tag{5}$$

The global minimum of the relaxed c–MDLR can be easily obtained using the Lagrange multipliers method [8], and is given by

$$\hat{k}_i = \frac{S + mR}{s_i} \frac{\sqrt{\lambda_i s_i w_i}}{\sum_{j=1}^{n} \sqrt{\lambda_j s_j w_j}}. \tag{6}$$

We truncate the values for $\hat{k}_i$ obtained by (6) so that they are all in the range $[1, m]$, and we call this solution the *Lagrange approximate* solution of the c–MDLR problem.

We also consider another approximation to the optimal of c–MDLR, which we call the *naive approximate* solution of the c–MDLR, which is given by

$$\hat{k}_i = \min \left\{ \max \left\{ \frac{\lambda_i w_i}{\sum_{j=1}^{n} \lambda_j w_j} \frac{S + mR}{s_i}, 1 \right\}, m \right\} \tag{7}$$

for $i = 1, 2, \ldots, n$. The naive approximate solution of the c–MDLR problem is clearly very simple to compute.

An optimal or approximate solution $\hat{k}_i$ of the c–MDLR problem needs to be converted into an integer solution that satisfies the two constraints of the MDLR problem, and at the same time provides a good approximation for MDLR. To this end, we use the PACKING routine in Figure 1. The PACKING routine works as follows. First it ensures that each object has at least one replica. Second, for all the objects in decreasing order of their $\hat{k}_i$, it assigns as many replicas to each object (but no more than m) as long as there is available space for replication.

```
PACKING (List of Objects $x$, Available Space $A$)
1   round $\hat{k}_i$ to an integer with value $1 \leq \hat{k}_i \leq m$
2   sort $x_i$ based on $\hat{k}_i$
3   foreach object $x_i$ do
4       while(($A \geq s_i$) and ($\hat{k}_i \leq m$)) do
5           increment $\hat{k}_i$
6           $A -= s_i$
7   foreach object $x_i$ do, $k_i = \hat{k}_i$
```

Figure 1. Packing empty space by increasing number of replicas.

It is quite straightforward to implement the naive and Lagrange approximation algorithms for the MDLR problem in a decentralized manner (details provided in the next section). We compare the approximations to the optimal solution of the MDLR problem, obtained by the packing of the Lagrange approximate and naive approximate solutions of c–MDLR in the next section.

Replica Placement and Request Distribution

Consider an object x_i that has a k_i replicas. Let $\pi(x_i, k_i)$ be a sequence of k_i disks that contain the replicas of x_i, which we call the *replica probe* sequence of x_i. The problem then is to determine the replica probe sequence for each object. Since the number of replicas of an object changes dynamically, the replica probe sequence also changes. The replica probe sequences affect the load imposed on the disks.

To achieve load balancing among the disks for an object x_i, each disk in the replica probe sequence $\pi(x_i, k_i)$ should be servicing approximately the same number of requests for x_i. A simple way to do that would be, for those disks, to serve the requests for x_i in a round–robin manner. Since all requests are seen by all the disks, this load distribution can be easily done, if each disk knows the replica probe sequence for each object it stores and sees the disk that serviced the last request for x_i.

To ensure that all the disks service approximately the same weighted number of requests, the replicas of the objects should be properly distributed among the disks. A simple way to distribute the objects among the disks is to use hashing. In fact, we would like to have a hash function H that given x_i and k_i returns a replica probe sequence $\pi(x_i, k_i)$ of length k_i. An open addressing hash function, with a hash table of size m (where each bucket of the hash table corresponds to a disk), can provide us with such replica probe sequences. Double–hashing is a preferred method since it alleviates primary and secondary clustering among the disks (buckets) [7].

However, since we would like to be able to add and remove disks dynamically, one needs to be careful so that the replica probe sequences do not change drastically when a disk is added or removed. Note that the size of the hash table assumed by the hash function is equal to the number of disks. Unfortunately, the probe sequences that one gets from double hashing could change quite drastically when the number of disks changes (especially for small number of replicas). Thus, double–hashing alone is insufficient. It seems that what is needed is a hash function that requires few objects to be transferred among buckets, when the size of its hash table changes by one. Linear hashing [10] has the property that when a bucket is added only half of the objects in one of the existing buckets (called its *sibling bucket*) need to be transferred to the newly added bucket . Similarly, when the last bucket is removed, its objects need to be transferred to its sibling bucket. However, linear hashing does not provide us with probe sequences.

We propose a combination of double–hashing (or any other open addressing hash function) with linear hashing for constructing replica probe sequences that enable good object placement and request distribution among the disks. Let h be a double–hashing hash function with a hash table whose size M is an upper bound on the number of disks a system will have. Let $h(x_i)$ be the probe sequence, of length M, for x_i returned by h. It is important that h gives probe sequences that utilize the whole space (which is achievable with double hashing [7]). Thus, we can assume without loss of generality, that the sequence $h(x_i)$ has M distinct entries.

Let ϕ_m be the linear hash function for a hash table of size m. We use this function to map an integer $0 \leq k \leq M-1$ into an integer (bucket, disk) in $0, 1, 2, \ldots, m-1$ as follows:

$$\phi_m(k) = \begin{cases} \hat{k}, & \text{if } \hat{k} < m \\ \hat{k} - 2^{\lceil \log_2 m \rceil - 1}, & \text{otherwise} \end{cases} \quad (8)$$

where $\hat{k} = k \bmod 2^{\lceil \log_2 m \rceil}$.

Let $\phi_m(x_i)$ be the sequence obtained by applying ϕ_m to each element of $h(x_i)$. If M is much larger than m, then $\phi_m(x_i)$ will have m unique entries with high probability, since we are placing the M distinct entries of $h(x_i)$ into the linear hash table of size m. We construct the replica probe sequence $\pi(x_i, k_i)$ for x_i by choosing the first k_i distinct elements of the sequence $\phi_m(x_i)$. As stated above, $\phi_m(x_i)$ has m distinct elements if $h(x_i)$ contains all the integers in $[0, M-1]$ and $M \geq m$. Thus, if $\forall x_i$, $h(x_i)$ contains $[0, M-1]$ and $M > m$ then the maximal replica probe sequence $\pi(x_i, m)$ is always defined. Further, we denote the first element of $\pi(x_i, k_i)$ to be the *primary disk* for x_i. It is easy to see that, when $k_i = 1$, the object x_i is stored at its corresponding primary disk. Figure 2 illustrates our method for constructing replica probe sequences.

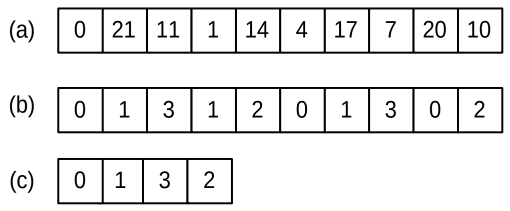


Figure 2. (a) the first 10 elements of the probe sequence of length M for x_{100}, using the hash function $h(x_i) =< z_0, z_2, \ldots, z_{M-1} >$, where $z_j = ((i \bmod M) + j(1 + i \bmod (M-1))) \bmod M$ and $M = 23$. (b) the first 10 elements of the sequence $\phi_m(x_{100})$, which is obtained by applying ϕ_m to each element of $h(x_{100})$ for $m = 4$. (c) the replica probe sequence $\pi(x_{100}, k_{100})$ when $k_{100} = m$.

Handling Addition and Removal of Disks

In this subsection, we show how the proposed system can adapt to addition and removal of disk(s). Assume that we are initially given m disks $D_0, D_1, \ldots, D_{m-1}$. When the number of disks change, each of the existing disks needs to recompute the replica probe sequences for all the objects x_i it stores. Moreover, from the property of linear hashing, it can be shown that the replica probe sequences of each x_i, before and after the change, differ by *at most two elements*. We use this property to guarantee that each disk in the new replica probe sequence of x_i, has a replica of x_i.

Adding a new disk D_m. For each object x_i it stores, D_m does the following: (a) if x_i is already available at another disk then D_m deletes its own replica of x_i; else, (b) if D_m has the unique copy of x_i in the system, it transfers x_i to the single disk in the new replica probe sequence of x_i (unless that disk is D_m). Each other disk D_j does the following for each object x_i it stores: D_j transfers its copy of x_i to D_m if the new replica probe sequence for x_i contains m but not j.

Removing a disk D_j. There are two cases to consider.

Case 1: $D_j = D_{m-1}$. Disk D_{m-1} first "*empties*" all the copies of objects x_i it stores as follows. If there exists a disk D_z which is in the new replica probe sequence of x_i (obtained after the deletion of D_j), but not in the old one, then D_{m-1} transfers its copy of x_i to disk D_z. Otherwise, D_{m-1} deletes its copy of x_i decreasing the number of replicas of x_i by one. Finally, disk D_{m-1} is removed from the system.

Case 2: $D_j \neq D_{m-1}$. We do the following: (a) disk D_{m-1} is emptied as in case 1 above, (b) all the (copies of) objects stored in D_j are transfered to D_{m-1}, and (c) disk D_{m-1} is renamed as D_j.

In both cases, disks may need to adjust their information about which disk serviced the last request for an object x_i: specifically, if that disk was D_{m-1}, then all other disks assume that the disk that precedes D_{m-1} in the replica probe sequence of x_i, serviced the last request for x_i.

Handling Disk Failures

We assume that a disk can be in one of these three states: operational, failed, or recovering. We assume that for each file/object, there is at least one operational disk in its replica probe sequence. A create operation succeeds if the single disk in the replica probe sequence of that object is operational. All other operations always succeed, since there is always an operational disk in the replica probe sequence of an object. Only operational disks can service requests. For the remainder, we consider, w.l.o.g., object read operations.

Disk D_j fails temporarily if it does not service a pending request for x_i within a timeout period when all the other operational disks in the replica probe sequence $\pi(x_i, k_i)$ expect it to do so. Disk D_j fails permanently if it does not recover within a certain timeout period.

A temporary failure of D_j is handled as follows. When the operational disks in the replica probe sequence $\pi(x_i, k_i)$ of x_i, detect the temporary failure of D_j, they assume that it is the turn of the disk that follows D_j in $\pi(x_i, k_i)$ to service that pending request for x_i. Further, a recovering disk D_j becomes operational (w.r.t to each object x_i it stores) upon completing the following recovery procedure. It issues a read request for x_i, and upon getting a response it either updates its replica of x_i, if it is still in $\pi(x_i, k_i)$, or deletes its own replica of x_i otherwise.

A permanent failure of D_j is handled as follows. Remove D_j as described above, but instead of transferring objects x_i from D_j, we transfer them from one of the operational disks in their replica probe sequences.

PERFORMANCE EVALUATION

We have developed a JAVA–based discrete event simulator for a Network–Attached Storage system to implement and evaluate our algorithms.

We consider a system of 16 disks that are connected to a high-capacity network. We configure the network so that it is not the bottleneck of the system and choose parameters that are similar to a 1 Gb/s Ethernet. Our disk model in Table 1 roughly matches the CMU–NASD system [9].

Table 1. Disk Parameters

Seek mean	5.4 ms
Seek max	11.0 ms
Rotation mean	2.99ms
Rotation max	5.98 ms
Transfer Rate	200 Mbits/second
On-board Processor	133 MHz

Table 2. File Size Distribution

Percentage	File Size (KB)
33	1
21	2
13	4
10	8
8	16
5	32
4	64
3	128
2	256
1	1000

There is a single client that issues requests for the objects, according to a synthetic workload. For the current experiments, we consider 10000 files/objects that are initially distributed among the disks. The size of the objects vary according to the SPECsfs 3.0 benchmark [1] as shown in Table 2. Further, the total disk space available for replication, as a fraction β, of the total size of all the objects in the system, is varied between 2.0 and 4.0. This implies that if all objects were treated equally, we can potentially have between 2 and 4 replicas for each object, respectively.

Clients request for one of the following operations on an object : read, write, get attribute and set attribute. We assume that the percentage of read, write, get attribute and set attribute operations are 67.1, 20.8, 11.8 and 0.3$\%$ respectively (loosely based on SPECsfs 3.0). Recent studies [5] suggest that the reference probability for Web documents requested by a client does not follow Zipf's Law precisely, but instead follows a Zipf-like distribution with the exponent varying from trace to trace. Specifically, if documents are ranked according to their access frequencies, then the reference probability for a document with rank i is proportional to $1/i^{\alpha}$. Note that for $\alpha = 1$, the request distribution strictly follows Zipf's law. However, as shown in [5], the distribution of Web requests follows a Zipf-like distribution, with $0.64 \leq \alpha \leq 0.83$. In our experiments, a client requests objects according to a Zipf-like distribution, where the object number corresponds to its rank, and the values of α used are 0.64, 0.75, and 0.83 respectively.

The inter-arrival time for client requests is assumed to be exponentially distributed. We simulate different load conditions on the system by varying the mean inter–arrival time between 0.01 seconds and 0.0005 seconds, thereby varying the client request rate between 100 and 2000 requests/second).

As described in previously, the objective function is to minimize the average *weighted* number of requests serviced by a disk, without exceeding the total space available for replication at the disks. For the rest of the section, we use the following terminology : OPT-A, OPT-B and OPT-C each refers to the packing of the optimal solution of the c-MDLR problem, where weight w_i assigned to each request for object x_i is equal to (a) 1, (b) s_i, and (c) $1 + \alpha s_i$, respectively. The Lagrange approximation algorithms LAGRANGE-A, LAGRANGE-B and LAGRANGE-C; and the naive approximation algorithms NAIVE-A, NAIVE-B and NAIVE-C are defined similarly. We next compare the performance of the approximation algorithms for the MDLR problem with different workloads.

Some Offline Results

For this set of experiments, we generate 25 different workloads for 10000 objects, each consisting of 10^6 requests. The size of objects and the size of disks are based on setup described above. For each workload, we fix $\beta = 3.0$ and obtain the approximate solutions of the MDLR problem from the packing of the optimal, naive and Lagrange approximate solutions to the c-MDLR problem. Note that, each solution instance gives the integral number of replicas k_i for each object x_i. We use the k_i values in equation (1) (with $w_i = 1$) to compute the average number of requests serviced by each disk. Table 3 shows the results given by each approximation algorithm, averaged over the 25 workloads.

Table 3. Average number of requests serviced by a disk, given by the packing of the optimal, naive approximation and Lagrange approximation solutions.

Method	Average Number of Requests
OPT-A	6602
OPT-B	6781
OPT-C	6493
NAIVE-A	8691
NAIVE-B	8710
NAIVE-C	8624
LAGRANGE-A	6606
LAGRANGE-B	6884
LAGRANGE-C	6552

Our results show that using weight $w_i = 1 + \alpha s_i$, gives the minimum average number of disk requests, for each of the approximation schemes. Moreover, the packing of the Lagrange approximation to c-MDLR, yields a solution that is always within 1% of that given by the packing of the optimal solution to c-MDLR. The optimal algorithm for the c-MDLR is both computationally expensive and difficult to implement in a decentralized manner. Thus, we focus on the relative performance of the naive and Lagrange approximations to MDLR, within the simulation framework.

Simulation Results

For a particular choice of an algorithm, the current implementation specifies that each disk D computes the required number of replicas of each object x_i, for which D is the primary (as defined earlier). In order to make it adaptive to changes in client access patterns, the replication algorithm is run periodically (every 10000 requests) at the primary disks, thus adjusting the number of replicas for some (or all) the objects. Subsequently, each disk can compute the replica probe sequences to check whether it should add or drop a replica of any object [2].

We now present our main results for $\alpha = 0.75$ and $\beta = 3.0$. Similar results were obtained for other settings of α and β , and are omitted due to lack of space.

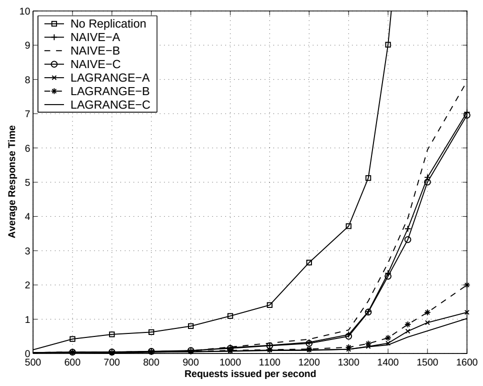


Figure 3. Average response time with increasing client request rates.

Figure 3 shows the average response time of the system, as the load, i.e. client request rate, is increased from 100 requests/second to 1600 requests/second. We use the average response time as a metric for evaluating the maximum load that the storage system can sustain. Our results show that, under low to medium load conditions, both the naive and Lagrange solutions perform well in terms of delivering low response time. However, the response time for the naive schemes start increasing at 1000 requests/second, and grow rapidly for load(s) in excess of 1200 requests/second. At similar loads, we observe that the Lagrange schemes do a good job in keeping the response time within acceptable

[2]With disk processing capabilities of 133 MHz, the double/linear hash functions can be computed by utilizing minimal fraction (about 0.8% according to[4]) of CPU time

values. Further, the average response time for the Lagrange schemes start increasing at 1400 requests/second, and grow rapidly for load(s) in excess of 1600 requests/second. Note that, from a queuing theory perspective, the maximum stable load for the system is about 1800 requests/second (using the $M/M/16$ model). We further note that LAGRANGE-C performs the best among all the algorithms, while both NAIVE-B and LAGRANGE-B perform worst in their respective categories.

Based on these observations, we can make a number of inferences: (a) the Lagrange schemes make the best utilization of the space available for replication and improve the system performance by about 30% when compared to the naive schemes, and by more than 100% when compared to no replication, (b) when calculating the number of replicas, it is a good idea to weigh each request for an object by a function that incorporates the disk seek time and the transfer time (rate) for the object, (c) it is *not* a good idea to weigh each request for an object by its size only, as this leads to more replicas being created for larger (but less popular) objects, thereby leading to poor utilization of the available storage.

For the rest of the section, we focus on the approximation algorithms NAIVE-C and LAGRANGE-C. Specifically, we look at the system throughput and performance of the disks at loads of 1200 and 1400 requests/second, respectively.

Fig. 4(a) shows the average response time for NAIVE-C and LAGRANGE-C as the simulation progresses. Observe that, LAGRANGE-C adapts to the demand far more quickly than NAIVE-C. Moreover, when both the schemes stabilize, the response time of NAIVE-C is about 4 times that of its Lagrange counterpart. From Fig. 4(b), we also observe that the average system throughput for LAGRANGE-C is about 50% more than NAIVE-C.

Figures 4(c)–(d) provide a better explanation of this improvement. Fig. 4(c) shows the maximum requests issued to any disk, as well as the median number of requests issued to a disk [3]. While NAIVE-C does a good job of minimizing the median number of requests issued to a disk, it fails to eliminate all the *hot spots*. Consequently, some of the disks continue to receive a large number of requests, thereby increasing the response time and degrading the system throughput. In contrast, LAGRANGE-C succeeds in equalizing the requests issued across the disks. The improvement is more significant for the maximum and median MBytes requested from the disks, as shown in Fig. 4(d). In this case, both the maximum and median values for NAIVE-C are well above the corresponding values for LAGRANGE-C. In summary , the Lagrange approximation scheme performs significantly better than the naive approximation scheme, in terms of utilizing the available storage and minimizing the average number of requests (and bytes) serviced by the disks.

Figures 5(a)–(d) show the same metrics when the load is increased to 1400 requests/second. In this case, the average response time for NAIVE-C starts growing rapidly, while the response time for LAGRANGE-C stabilizes at under 0.05 seconds. The system throughput of LAGRANGE-C is also about 50% more than NAIVE-C. Once again, as shown in Figs. 5(c)–(d), the Lagrange approximations does a far better job in balancing the requests (bytes) among the disks, when compared to the naive approach.

CONCLUSIONS

The design of an efficient and scalable storage system is a critical aspect for the performance of current data–intensive network services. In this paper, we propose a simple and novel approach for data (object) replication and request distribution in network–attached storage systems. Replica calculations are based on an approximate solution to the MDLR (Minimum Disk Load Replication) problem using the Lagrange multipliers method. Further, we demonstrate how the replication algorithm can be combined with hashing to (a) balance load among the disks, and (b) adapt to changes in the operating environment in an efficient manner. Experiments with synthetic workloads show that, under demanding conditions, our algorithm largely outperforms naive alternatives in terms of response time and throughput.

References

[1] Standard Performance Evaluation Corp. SPECsfs97 V3.0.

[2] Seagate Technology Inc. Object Oriented Devices: Description of Requirements, 1998.

[3] Amiri, K. S. Scalable and Manageable Storage Systems. *Ph.D. Dissertation, CMU-CS-00-178*, 2000.

[4] Anderson, D., Chase, J., and Vahdat, A. Interposed Request Routing for Scalable Network Storage. In *Proceedings of OSDI*, 2000.

[5] Breslau, L., Cao, P., Fan, L., Philips, G., and Shenker, S. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE Infocom*, March 1999.

[6] Brown, A., Oppenheimer, D., Keeton, K., Thomas, R., Kubiatowicz, J., and Patterson, D. ISTORE: Introspective Storage for Data-Intensive Network Services. In *Proceedings of HotOS-VII*, 1999.

[7] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1998.

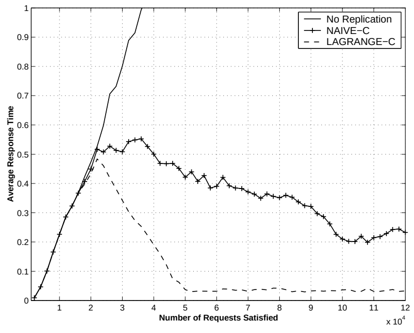
[8] Fletcher, R. *Practical Methods of Optimization, Second Edition*. John Wiley and Sons Ltd., Reading, MA, USA, 1990.

[9] Gibson, G., Nagle, D., Nat, W., Paul, L., Marc, M., and Jim, U. NASD scalable storage systems. In *Proceedings of USENIX*, 1999.

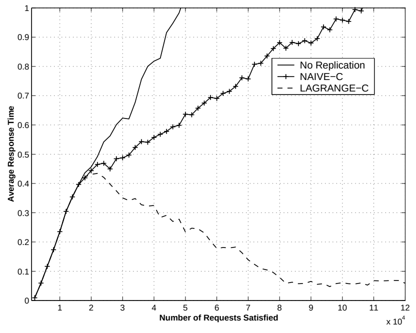
[10] Litwin, W. Linear hashing: A new tool for file and table addressing. In *Proceedings of VLDB*, 1980.

[11] Thekkath, C. A., Mann, T., and Lee, E. K. Frangipani: A Scalable Distributed File System. In *Proceedings of Symposium on Operating Systems Principles*, 1997.

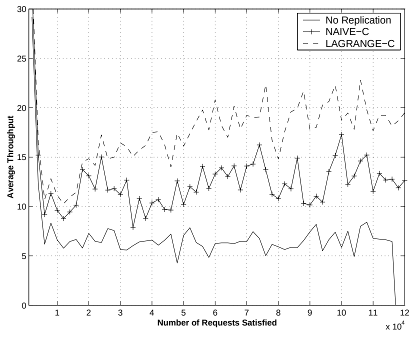
[3] We use the median since it is a more robust measure in cases of high variability.



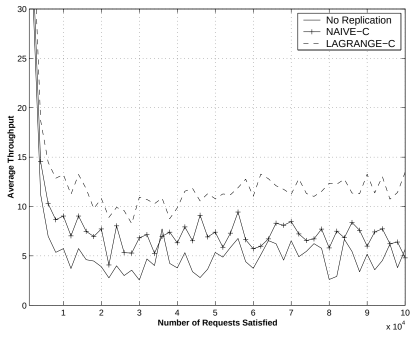
(a)Average Response Time (seconds)



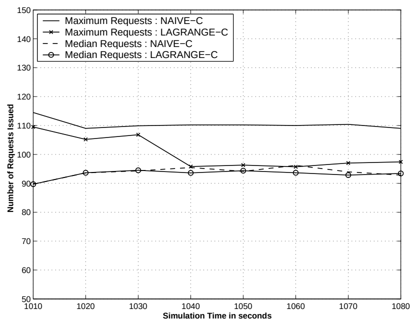
(a)Average Response Time (seconds)



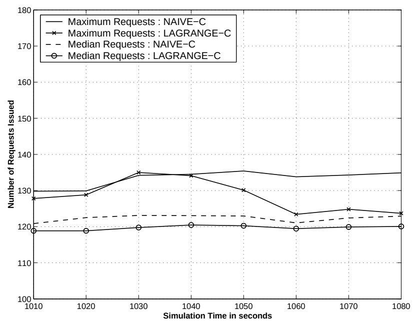
(b)Average Throughput (MBytes/second)



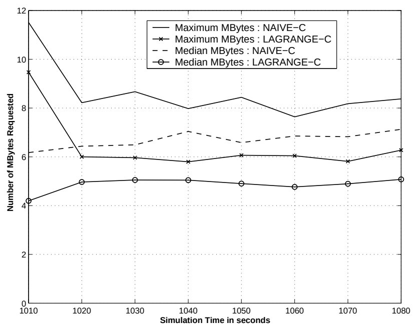
(b)Average Throughput (MBytes/second)



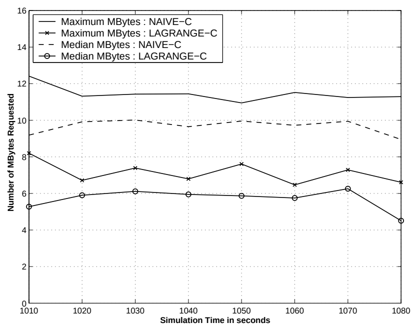
(c)Maximum/Median Requests issued to a disk



(c)Maximum/Median Requests issued to a disk



(d)Maximum/Median MBytes requested from a disk



(d)Maximum/Median MBytes requested from a disk

Figure 4. Performance measures for request rate of 1200 requests/second.

Figure 5. Performance measures for request rate of 1400 requests/second.