A Fast Algorithm for Online Placement and Reorganization of Replicated Data

R. J. Honicky
Storage Systems Research Center
University of California, Santa Cruz

Ethan L. Miller
Storage Systems Research Center
University of California, Santa Cruz

Abstract

As storage systems scale to thousands of disks, data distribution and load balancing become increasingly important. We present an algorithm for allocating data objects to disks as a system as it grows from a few disks to hundreds or thousands. A client using our algorithm can locate a data object in microseconds without consulting a central server or maintaining a full mapping of objects or buckets to disks. Despite requiring little global configuration data, our algorithm is probabilistically optimal in both distributing data evenly and minimizing data movement when new storage is added to the system. Moreover, our algorithm supports weighted allocation and variable levels of object replication, both of which are needed to permit systems to efficiently grow while accommodating new technology.

1 Introduction

As the prevalence of large distributed systems and clusters of commodity machines has grown, significant research has been devoted toward designing scalable distributed storage systems. Scalability for such systems has typically been limited to allowing the construction of a very large system in a single step, rather than the slow accretion over time of components into a large system. This bias is reflected in techniques for ensuring data distribution and reliability that assume the entire system configuration is known when each object is first written to a disk. In modern storage systems, however, configuration changes over time as new disks are added to supply needed capacity or bandwidth.

The increasing popularity of network-attached storage devices (NASDs) [11], which allow the use of thousands of "smart disks" directly attached to the network, has complicated storage system design. In NASD-based systems, disks may be added by connecting them to the network, but efficiently utilizing the additional storage may be difficult. Such systems cannot rely on central servers because doing so would introduce scalability and reliability problems. It is also impossible for each client to maintain detailed information about the entire system because of the number of devices involved.

Our research addresses this problem by providing an algorithm for a client to map any object to a disk using a small amount of infrequently-updated information. Our algorithm distributes objects to disks evenly, redistributing as few objects as possible when new disks are added to preserve this even distribution. Our algorithm is very fast, and scales with the number of disk groups added to the system. For example, a 1000 disk system in which disks were added ten at a time would run in time proportional to 100. In such a system, a modern client would require about 10 μs to map an object to a disk. Because there is no central directory, clients can do this computation in parallel, allowing thousands of clients to access thousands of disks simultaneously.

Our algorithm also enables the construction of highly reliable systems. Objects may have an arbitrary, adjustable degree of replication, allowing storage systems to replicate data sufficiently to reduce the risk of data loss. Replicas are distributed evenly to all of the disks in the system, so the load from a failed disk is distributed evenly to all other disks in the system. As a result, there is little performance loss when a large system loses one or two disks.

Even with all of these benefits, our algorithm is simple. It requires fewer than 100 lines of C code, reducing the likelihood that a bug will cause an object to be mapped to the wrong server. Each client need only keep a table of all of the servers in the system, storing the network address and a few bytes of additional information for each server. In a system with thousands of clients, a small, simple distribution mechanism is a big advantage.

2 Related Work

Litwin, *et al.* describe a class of data structures and algorithms on those data structures which the authors dubbed Scalable Distributed Data Structures (SDDS) [20]. There are three main properties which a data structure must meet in order to be considered a SDDS.

1. A file expands to new servers gracefully, and only when servers already used are efficiently loaded.

This paper appeared at the *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003, Nice, France.

2. There is no master site that object address computations must go through, *e. g.*, a centralized directory.

3. File access and maintenance primitives, *e. g.*, search, insertion, split, etc., never require atomic updates to multiple clients.

While the second and third properties are clearly important for highly scalable data structures designed to place objects over hundreds or thousands of disks, the first property, as it stands, could be considered a limitation. In essence, a file that expands to new servers based on storage demands rather than on resource availability will present a very difficult administration problem.

Often, an administrator wants to add disks to a storage cluster and immediately rebalance the objects in the cluster to take advantage of the new disks for increased parallelism. An administrator does not want to wait for the system to decide to take advantage of the new resources based on algorithmic characteristics and parameters that they do not understand. This is a fundamental flaw in all of the LH* variants discussed below.

Furthermore, Linear Hashing and LH* variants split buckets (disks in this case) in half, so that on average, half of the objects on a split disk will be moved to a new, empty, disk. Moving half of the objects from one disk to another causes wide differences in the number of objects stored on different disks in the cluster, and results in suboptimal disk utilization [2]. Splitting in LH* will also result in a "hot spot" of disk and network activity between the splitting node and the recipient. Our algorithm, on the other hand, always moves a statistically optimal number of objects from every disk in the system to each new disk, rather than from one disk to one disk.

LH* variants such as LH*M [19], LH*G [21], LH*S [18], LH*SA [17], and LH*RS [22] describe techniques for increasing availability of data or storage efficiency by using mirroring, striping and checksums, Reed-Solomon codes and other standard techniques in conjunction with the basic LH* algorithm. Our algorithm can also easily take advantage of these standard techniques, although that is not the focus of this paper.

The LH* variants do not provide a mechanism for weighting different disks to take advantage of disks with heterogeneous capacity of throughput. This is a reasonable requirement for storage clusters which grow over time; we always want to add the highest performance or highest capacity disks to our cluster. Our algorithm allows weighting of disks. Breitbart, *et al.* [2] discuss a distributed file organization which resolves the issues of disk utilization (load) in LH*. They do not, however, propose any solution for data replication.

Kröll and Widmayer [14] propose another SDDS that they call Distributed Random Trees (DRTs). DRTs are optimized for more complex queries such as range queries and and closest match, rather than the simple primary key lookup supported by our algorithm and LH*. Additionally, DRTs support server weighting. Because they are SDDS's, however, they have the same difficulties with data-driven reorganization (as opposed to administrator-driven reorganization) as do LH* variants. In addition, the authors present no algorithm for data replication, although meta-data replication is discussed extensively. Finally, although they provide no statements regarding the average case performance of their data structure, DRT has worst-case performance which is linear in the the number of disks in the cluster. In another paper, the authors prove a lower bound of $\Omega(\sqrt{m})$ on the average case performance of any tree based SDDS [15], where m is the number of objects stored by the system. Our algorithm has performance which is $O(n \log n)$ in the number of groups of disks added; if disks are added in large groups, as is often the case, then performance will be nearly constant time.

Brinkmann, *et al.* [3, 4] propose a method for pseudo-random distribution of data to multiple disks using partitioning of the unit range. This method accommodates growth of the collection of disks by repartitioning the range and relocating data to rebalance the load. However, this method does not allow for the placement of replicas, an essential feature for modern scalable storage systems.

Chau and Fu discuss and propose algorithms for declustered RAID whose performance degrades gracefully with failures [5]. Our algorithm exhibits similarly graceful degradation of performance: the pseudo-random distribution of objects (declustering) means that the load on the system is distributed evenly when a disk fails.

Peer-to-peer systems such as CFS [10], PAST [24], Gnutella [23], and FreeNet [7] assume that storage nodes are extremely unreliable. Consequently, data has a very high degree of replication. Furthermore, most of these systems make no attempt to guarantee long term persistence of stored objects. In some cases, objects may be "garbage collected" at any time by users who no longer want to store particular objects on their node, and in others, objects which are seldom used are automatically discarded. Because of the unreliability of individual nodes, these systems use replication for high availability, and are less concerned with maintaining balanced performance across the entire system.

Other large scale persistent storage systems such as Farsite [1] and OceanStore [16] provide more file system-like semantics. Objects placed in the file system are guaranteed (within some probability of failure) to remain in the file system until they are explicitly removed (if removal is supported). OceanStore guarantees reliability by a very high degree of replication. The inefficiencies which are introduced by the peer-to-peer and wide area storage systems address security, reliability in the face of highly unstable nodes, and client mobility (among other things). These features introduce far too much overhead for a tightly coupled mass object storage system.

Distributed file systems such as AFS [13] use a client server model. These systems typically use replication at each storage node, such as RAID [6], as well as client caching to achieve reliability. Scaling is typically done by adding volumes as demand for capacity grows. This strategy for scaling can result in very poor load balancing, and requires too much maintenance for large disk arrays. In addition, it does not solve the problem of balancing object placement.

3 Object Placement Algorithm

We have developed an object placement algorithm that organizes data optimally over a system of disks or servers while allowing online reorganization in order to take advantage of newly available resources. The algorithm allows replication to be determined on a per-object basis, and permits weighting to distribute objects unevenly to best utilize different performance characteristics for different servers in the system. The algorithm is completely decentralized and has very minimal storage overhead and minimal computational requirements.

3.1 Object-based Storage Systems

NASD-based storage systems are built from large numbers of relatively small disks attached to a high bandwidth network, as shown in Figure 1. Often, NASD disks manage their own storage allocation, allowing clients to store *objects* rather than blocks on the disks. Objects can be any size and may have any 64-bit name, allowing the disk to store an object anywhere it can find space. If the object name space is partitioned among the clients, several clients can store different objects on a single disk without the need for distributed locking. In contrast, blocks must be a fixed size and must be stored at a particular location on disk, requiring the use of a distributed locking scheme to control allocation. NASD devices that support an object interface are called *object-based storage devices* (OBSDs)[1] [25]. We assume that the storage system on which our algorithm runs is built from OBSDs.

Our discussion of the algorithm assumes that each object can be mapped to a key x. While each object must have a unique identifier in the system, the key used for our algorithm need not be unique for each object. Instead, objects are mapped to a "set" that may contain hundreds or thousands of objects, all of which share the key x while having different identifiers. Once the algorithm has located the set in which an object resides, that set may be searched for the desired object; this search can be done locally on the OBSD and the object returned to the client. By restricting the magnitude of x to a relatively small number, perhaps 10^6 or 10^7, we make the object balancing described in Section 6.1 simpler to implement without losing the desirable balancing characteristics of the algorithm.

[1]OBSDs may also be called *object-based disks* (OBDs).

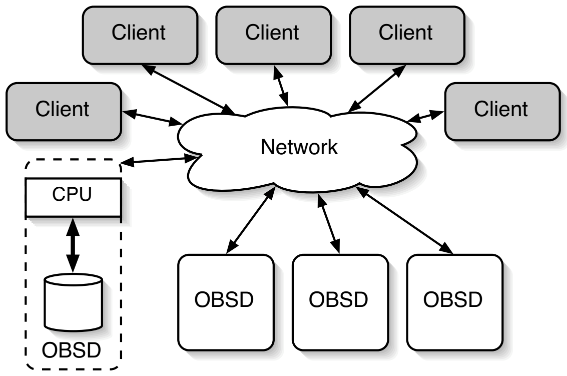


Figure 1. A typical NASD-based storage system

Most previous work has either assumed that storage is static, or that storage is added for additional capacity. We believe that additional storage will be necessary as much for additional performance as for capacity, requiring that objects be redistributed to new disks. If objects are not rebalanced when storage is added, newly created objects will be more likely to be stored on new disks. Since new objects are more likely to be referenced, this will leave the existing disks underutilized.

We assume that disks are added to the system in *clusters*, with the jth cluster of disks containing m_j disks. If a system contains N objects and $n_j = \sum_{i=0}^{j-1} m_i$ disks, adding m more disks will require that we relocate $N \cdot \frac{m}{n_j+m}$ objects to the new disks to preserve the balanced load. For all of our algorithms, we assume that existing clusters are numbered $0 \ldots c-1$, and that we are adding cluster c. The cth cluster contains m_c disks, with n_c disks already in the system.

3.2 Basic Algorithm

We will call disks *servers* since this algorithm might be used to distribute data in an object database or other more complex service. Our algorithm operates on the basic principle that in order to move the (statistically) optimal number of objects into a new cluster of servers, we can simply pick a pseudo-random integer $z_x = f(x)$ based on each object's key x such that $0 \le z_x < n_c + m_c$. If $z_x < m_c$, then the object in question moves to the new cluster. Our algorithm is applied recursively; each time we add a new cluster of servers, we

```
$j = c$
while (object not mapped)
  seed a random number generator with the object's key $x$
  advance the random number generator $j$ steps.
  generate a random number $0 \leq z < (n_j + m_j)$
  if $z \geq m_j$
    $j \leftarrow j - 1$
  else
    map the object to server $n_j + (z \mod m_j)$
```

Figure 2. Algorithm for mapping objects to servers without replication or weighting.

add another step in the lookup process. To find a particular object, we work backward through the clusters, starting at the most recently added, deciding whether the object would have been moved to that cluster. The basic algorithm for determining the placement of some object with key x, before making considerations for object replication, and weighting is shown in Figure 2.

We use a uniform random number generator which allows "jump-ahead": the next s numbers generated by the generator can be skipped, and the $s+1$st number can be generated directly. The generator which we use can be advanced s steps in $O(\log s)$ time, but we are currently exploring generators which can generate parametric random numbers in $O(1)$ time, as described in Section 5.1.

Using a simple induction, we sketch a proof that the expected number of objects placed in the new cluster by this basic algorithm is $\frac{m_c}{n_c+m_c} \cdot N$, and that objects will be randomly distributed uniformly over all of the servers after the reorganization. We also demonstrate that the algorithm minimizes the expected number of objects which get moved in a reorganization where only a single cluster is added, and that the algorithm is therefore optimal in the number of objects moved during such a reorganization.

In the base case, all objects should clearly go to the first cluster since $n_0 = 0$, meaning that $\frac{m_0}{n_0+m_0} \cdot N = N$. Furthermore, since z comes from a uniform distribution and each object will be placed on server $0 + (z \mod m_0) = z \mod m_0$, the probability of choosing a given server is $\frac{1}{m_0}$. Thus each server has an equal probability of being chosen, so the objects will be distributed uniformly over all of the servers after placing them on the first cluster.

For the induction step, assume that N objects are randomly distributed uniformly over n_c servers divided into $c-1$ clusters, and we add cluster c containing m_c servers. We will optimally place $\frac{m_c}{n_c+m_c} \cdot N$ objects in cluster c.

Since each random number $0 \leq z < n_c + m_c$ is equally likely, we have a probability of $\frac{m_c}{n_c+m_c}$ of moving any given object to a server in cluster c. With N objects, the total number of objects moved to a server in cluster c is $\frac{m_c}{n_c+m_c} \cdot N$—the optimal value.

Since the N objects in the system are distributed uniformly over n_c servers by our inductive hypothesis, a relocated object has an equal probability of coming from any of n_c servers. The expected number of objects moved from any given server S (where $0 \leq S < n_c$) is $\frac{m_c}{n_c+m_c} \cdot \frac{1}{n_c} \cdot N$. so the expected number of objects remaining on any server S will be $\frac{1}{n_c}\left(1 - \frac{m_c}{n_c+m_c}\right) \cdot N = \frac{N}{n_c+m_c}$. Since the expected number of objects placed in cluster c is $\frac{m_c}{n_c+m_c} \cdot N$, the expected number of objects placed on a given server in cluster c is $\frac{1}{m_c} \cdot \frac{m_c}{n_c+m_c} \cdot N = \frac{N}{n_c+m_c}$.

Because the expected number of objects on any server in the system after reorganization is $\frac{N}{n_c+m_c}$, the distribution of objects in the system remains uniform. Since the decision regarding which objects to move and where to move them is made using a pseudo-random process, the distribution of objects in the system also remains random.

Finally, we can see that the algorithm moves an approximately optimal number of objects during the reorganization by noting two facts. First, an object mapped to a given cluster will never move to a different cluster unless it is mapped to a newly added cluster—objects may move to new clusters, but never to old ones. When we add a new cluster, all objects that move must therefore move into the new cluster. Secondly, the expected number of objects in a new cluster is exactly the number of objects which will allow the distribution of objects over the clusters to remain uniform, so the algorithm could not move fewer objects into the new cluster and remain correct. We therefore move approximately the minimum number of objects for the algorithm to remain correct. Therefore, the algorithm moves the optimal number of objects during a reorganization.

4 Cluster Weighting and Replication

Simply distributing objects to uniform clusters is not sufficient for large-scale storage systems. In practice, large clusters of disks will require weighting to allow newer disks (which are likely to be faster and larger) to contain a higher proportion of objects than existing servers. Such clusters will also need replication to overcome the frequent disk failures that will occur in systems with thousands of servers.

4.1 Cluster Weighting

In most systems, clusters of servers have different properties—newer servers are faster and have more capacity. We must therefore add weighting to the algorithm to allow some server clusters to contain a higher proportion of objects than others. To accomplish this, we use a integer weight adjustment factor w_j for every cluster j. This factor will likely be a number which describes the power (such as capacity, throughput, or some combination of the two) of the server. For example, if clusters are weighted by the capacity of the drives, and each drive in the first cluster is 60 gigabytes, and each drive in the second cluster is 100 gigabytes, then w_0 might be initialized to 60, and w_1 might be initialized to 100. We then use $m'_j = m_j w_j$ in place of m_j and $n'_j = \sum_{i=0}^{j-1} m'_i$ in place of n_j in Figure 2. Once an object's cluster has been selected, it can be mapped to a server by $n_j + v \mod m_j$, as done in the basic algorithm.

The use of 64-bit integers and arithmetic allows for very large systems; a 1,000 terabyte system that weights by gigabytes will have a total weight of only 1 million. If weights are naturally fractional (as for bandwidth, perhaps), they can all be scaled by a constant factor c_w to ensure that all w_j remain integers.

4.2 Replication

The algorithm becomes slightly more complicated when we add replication because we must guarantee that no two replicas of an object are placed on the same server, while still allowing the optimal placement and migration of objects to new server clusters.

This version of the algorithm, shown in Figure 3, relies on the fact that multiplying some number $0 \le n < m$ by a prime p which is larger than m and taking the modulus m (*i. e.*. $(np) \mod m$) defines a bijection between the ordered set $S = \{0 \ldots m-1\}$ and some permutation of S [9]. Furthermore, the number of unique bijections is equal to the number of elements of S which are relatively prime to m. In other words, multiplying by a prime larger than m permutes the elements of S in one of $\phi(m)$ ways, where $\phi(\cdot)$ is the Euler Phi function [9], as described in Section 4.3.

Again, x is the key of the object being placed, c is the number of clusters, n_j is the total number of servers in the first $j-1$ clusters, and m_j is the number of servers in cluster j, where $j \in \{0 \ldots c-1\}$. Let R equal the maximum degree of replication for an object, and $r \in \{0 \ldots R-1\}$ be the replica number of the object in question. z and s are pseudo-random values used by the algorithm.

The algorithm also assumes that $m_0 \ge R$. That is, the number of servers in the first cluster is at least as large as the maximum degree of replication. This makes intuitive sense since if it were not true, there would not be a sufficient number of servers available to accommodate all of the replicas of an object when the system is first brought online.

In the case where $m_j < R$, our algorithm (intuitively speaking) first pretends that the cluster is of size R. It then selects only those object replicas which would be allocated to the first m_j servers in our imaginary cluster or R servers. In this way, we can avoid mapping more than one replica to the same server. When $m_j < R$, the number of objects

```
$j \leftarrow c$
**while** object is not mapped
  seed a random number generator with the object's key $x$
  advance the generator $j$ steps
  $m'_j \leftarrow m_j w_j$
  $n'_j \leftarrow \sum_{i=0}^{j-1} m'_i$
  generate a random number $0 \le z < (n'_j + m'_j)$
  choose a random prime number $p \ge m'_j$
  $v \leftarrow x + z + r \times p$
  $z' \leftarrow (z + r \times p) \mod (n'_j + m'_j)$
  **if** $m_j \ge R$ **and** $z' < m'_j$
    map the object to server $n_j + (v \mod m_j)$
  **else if** $m_j < R$ **and** $z' < R \cdot w_j$ **and** $v \mod R < m_j$
    map the object to server $n_j + (v \mod R)$.
  **else**
    $j \leftarrow j - 1$
```

Figure 3. Algorithm for mapping objects to servers with replication and weighting.

which get mapped into cluster j is $\frac{w_j \cdot R}{n'_j + m'_j} \cdot \frac{m_j}{R} = \frac{m'_j}{n'_j + m'_j}$, so the R factor cancels completely.

Let the total weight in the system W be $\sum_{i=0}^{c} w_i m_i$. The fraction of the total weight possessed by a server in cluster j is thus $\frac{w_i}{W}$. We must therefore show that the expected number of object replicas owned by some server j is $\frac{w_j}{W} \cdot N \cdot R$.

We also must show that no two replicas of the same object get placed on the same server. Again, we can prove these facts using induction. We omit the proof that the objects remain distributed over the other clusters according to their weights, since the argument is essentially identical to that in the basic algorithm described in Section 3.2.

In the base case, $n'_0 = 0$, and z' is modulus $n'_0 + m'_0 = m'_0$ (and hence $z' < m'_0$). Since we require that the first cluster have at least R servers, we will always map the object to server $n_0 + (v \mod m_0) = v \mod m_0$ which is in the first cluster, as described in Figure 3. v is a pseudo-random number (because z is pseudo-random), so an object has equal probability of being placed on any of the m_0 servers in cluster 0. Therefore, the expected number of objects placed on a given server when there is only one cluster is $\frac{1}{m_0} \cdot N \cdot R = \frac{w_0}{w_0 m_0} \cdot N \cdot R = \frac{w_0}{W} \cdot N \cdot R$, which is what we wanted to prove.

Now,

$$[x + z + r \times p] \equiv_{m_0} [x + z] + [r \times p].$$

We can therefore examine the $(x+z) \mod m_0$ term, and the $(r \times p) \mod m_0$ term separately.

Recall that x is the key of an object. Since x and z can be any value, both of which are (potentially) different for each object, but the same for each replica of the object, $x+z$ can

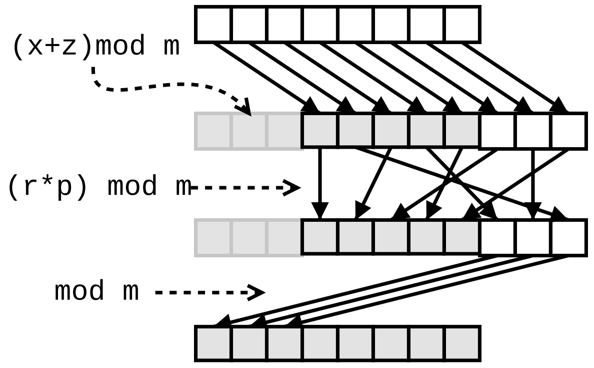


Figure 4. The mapping of the ordered set of integers $\{0,\ldots,m_j-1\}$ to a permutation of that set using the function $f(x) = (x+z+r\times p) \mod m_j$

be viewed as defining a random offset within the m_0 servers in the first cluster from which to start placing objects.

p and m_0 are relatively prime, so by the Chinese Remainder Theorem [9], for a given y, $[r\times p] \equiv_{m_0} y$ has a unique solution modulo m_0. In other words, p defines a bijection from the ordered set $\{0,\ldots,m_0-1\}$ to some permutation of that set.

Thus we can think of $(x+z+r\times p) \mod m_0$ as denoting some permutation of the set $\{0,\ldots,m_0-1\}$, shifted by $(x+z) \mod m_0$.[2] In other words, if we rotate the the last element to the first position $x+z$ times, then we have the set defined by $f(x) = (x+z+r\times p) \mod m_0$. Since this is also a permutation of $\{0,\ldots,m_0-1\}$, and since $r < m_0$, each replica of an object maps to a unique server, as shown in Figure 4.

For the induction step, assume that each cluster is weighted by some per-server (unnormalized) weight w_j where $0 \le j < c$, and that all of the object replicas in the system are distributed randomly over all of the servers according to each server's respective weight (defined by the server's cluster).

If we add a cluster c containing m_c servers, then $w_c \cdot m_c$ is the total weight allocated to cluster c. Since a given object replica is placed in cluster c with probability $\frac{w_c\cdot m_c}{W}$, the expected number of objects placed in cluster c is $\frac{w_c m_c}{W}\cdot N\cdot R$. As in the base case, the object replicas will be distributed over the servers in cluster c uniformly, so the expected number of object replicas allocated to a server in cluster c is $\frac{w_c}{W}\cdot N\cdot R$, which is what we wanted to show.

Since p defines a bijection between the ordered set $\{0,\ldots,m_c-1\}$ and some permutation of that set, each replica that is placed in cluster c is placed on a unique server. Note that at most m_c out of R replicas of a given object can be placed in cluster c, since the other $R-m_c$ replicas will be mapped mod R to values which are greater than or equal to m_c when $m_c < R$.

Thus, no two replicas of the same object get placed on the same server. Furthermore, following the same argument as given in Section 3.2 (omitted here for the sake of brevity), the algorithm moves (approximately) the optimal number of objects during a reorganization where a single cluster is added.

4.3 Choosing Prime Numbers

Our algorithm uses a random prime number, which must be known by every server and client in the system. It is sufficient to choose a random prime from a large pool of primes. This prime p will be relatively prime to any modulus $m < p$, as will $p \mod m$. Furthermore, choosing a random prime and computing $p \mod m$ is statistically equivalent to making a uniform draw from the set of integers in the set $\mathbb{Z}_m^* = \{0 \le x < m | \gcd(x,m) = 1\}$ which are relatively prime to m. A proof of this is beyond the scope of this paper.

The number of integers in the set $\mathbb{Z}_m^*$ (these relatively prime integers will be called *coprimes* for the remainder of this section) is described by the Euler Phi Function:

$$\phi(m) = m\prod_{p|m}\frac{p-1}{p}$$

where $p|m$ means the set of all p such that p is a factor of m[9].

Since $\phi(m) < m!$ when $m > 2$, the number of bijections described by the set of coprimes to m is smaller in general than the number of possible permutations of a set of integers $\{0,\ldots,m-1\}$. It is also beyond the scope of this paper to show the precise statistical impact of this difference. The practical impact of this difference, however, can be seen in Figure 6(c).

5 Performance and Operating Characteristics

5.1 Theoretical Complexity

In this section we demonstrate that our algorithm has time complexity of $O(nr)$ where n is the number of server additions made, and r is the time in which it takes to generate an appropriate random number. The algorithm that we are currently using to generate random numbers takes $O(\log n)$ time. This can theoretically be reduced to $O(1)$.

As noted in Section 4.3, appropriate prime numbers can be chosen in $O(1)$ time, and the rest of the operations other

[2]The number of unique permutations of $\{0,\ldots,m_0-1\}$ which can be obtained by multiplying by a coprime of m_0 is equal to the Euler Phi Function $\phi(m_0)$, as described in Section 4.3.

than those related to generating random numbers are arithmetic, so every operation besides those used for generating random numbers runs in $O(1)$ time.

The algorithm for seeding and actually generating random numbers is also constant time [26]. The algorithm for "jumpahead," or advancing the random number generator a given number of steps without calculating intermediate values, however takes $O(\log n)$ time. Specifically, the algorithm for jumpahead requires modular exponentiation, which is known to run in $O(\log n)$ time [9]. Since we must jump ahead by the cluster group number each iteration, each iteration of the algorithm takes, on average $O(\log n)$ time.

In the worst case, an object replica will be placed in the first server cluster, in which case the algorithm must examine every cluster to determine where the object belongs. The average case depends on the size and weighting of the different clusters, and thus is not a good metric for performance. If the weight and clusters sizes are distributed evenly, then clearly we will need, on average $\frac{n}{2}$ iterations. However, we believe that newer clusters will tend to have exponentially higher weights, so that in the average case, we only need to calculate $\log n$ iterations.

Rather than using jumpahead to generate statistically random hash values that are parameterized by the server cluster number, we have examined another approach using parametric random number generators [8]. These random number generators are popular for distributed random number generation. By parameterizing the generated sequence, the generators can assign a different parameter to each processor in a cluster, while using the same seed. This guarantees unique, deterministic pseudo-random number sequences for each processor.

One simple method, based on Linear Congruence Generators [8], allows the parameterization to occur in $O(1)$ time. LCGs, however, are notorious for generating numbers which all lie on a higher dimensional hyperplane, and thus are strongly correlated for some purposes. Unfortunately, this correlation results in very poor distribution of objects in our algorithm, making LCGs unusable for object distribution.

We are currently examining other more sophisticated generators, but as a final note, our algorithm does actually support $O(n)$ operation, but this is mostly of theoretical interest. $O(n)$ operation can be achieved as follows: On the first iteration, seed the generator and advance it n steps, as would normally be done. Next instead of re-seeding the generator and advancing it $n-1$ steps, retain the state of the generator (do not reseed it), and then advance it the period of the generator (in this case, the maximum value of an unsigned long integer) minus 1. Since the period of the generator is a known quantity which does not depend on n, this can be done in $O(1)$ time. Of course, advancing the generator by such a large quantity is very slow, so the classification as $O(n)$ is of academic interest only.

5.2 Performance

In order to understand the real world performance of our algorithm, we tested the average time per lookup under many different configurations. First, we ran a test in which 40,000 object replicas were placed into configurations starting with 10 servers in a single cluster to isolate the effect of server addition. We computed the average time for these 40000 lookups, and then added clusters of servers, 10 servers at a time, and timed the same 40,000 lookups over the new server organization. In Figure 5(a), we can see that the line for lookups under this configuration grows faster than linear, but much slower than $n \log n$.

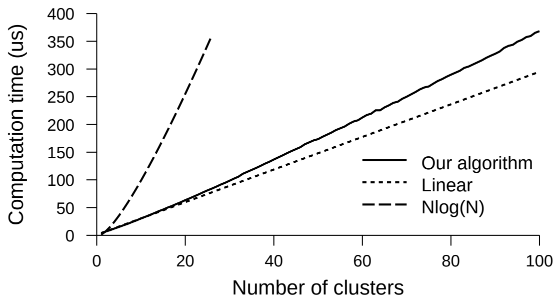
In Figure 5(b), there are two lines which grow approximately logarithmically. Since disk capacity has been growing exponentially [12], we also consider the performance of the algorithm when the weight of (and hence number of object assigned to) new clusters grows exponentially. The bottom line illustrates a 5% growth in capacity between cluster additions, and the middle line represents a 1% growth.

The weighting of new servers can therefore significantly improve the performance of the algorithm. This is consistent with the predictions made in Section 5.1.

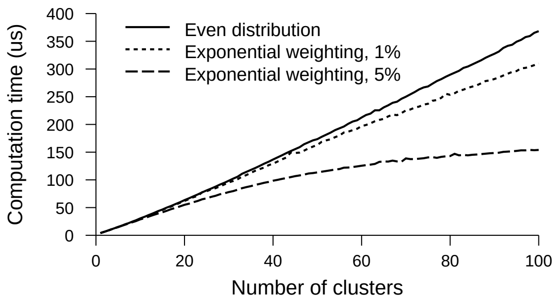
5.3 Failure Resilience

When a server fails, clients must read and write to other servers for each of the objects stored on the failed server. If all of the replicas for a particular server are all stored on the same set of servers, *e. g.* if all of the replicas for objects on server 3 are stored on server 4 and server 5, then a server failure will cause the read load on the "mirror servers" to increase by a factor of $\frac{R-1}{R}$, where R is the degree of object replication (meaning that the load on each of the mirror severs nearly doubles). This value assumes that the replicated clients are not using quorums for reads, in which case, all mirrors participate in reads, so that there will be no increase in load. This is a false benefit however, since it is achieved by using resources inefficiently during normal operation; $\frac{R-1}{R}$ can be a severe burden when R is 2–3, as likely will be used in large-scale systems. In order to minimize the load on servers during a failure, our algorithm places replicas of objects pseudo-randomly, so that when a server fails, the load on the failed server is absorbed by every other server in the system.

Figure 6(a) shows a histogram of the distribution of objects which replicate objects on server 6. In this case the load is very uniform, as it is in Figure 6(a), where the weight of each server cluster increases. In Figure 6(c), we see several spikes, and several servers which have no replicas of

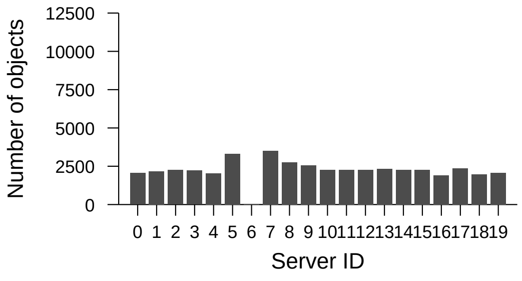


(a) Time per lookup compared to linear and $n \log n$ functions

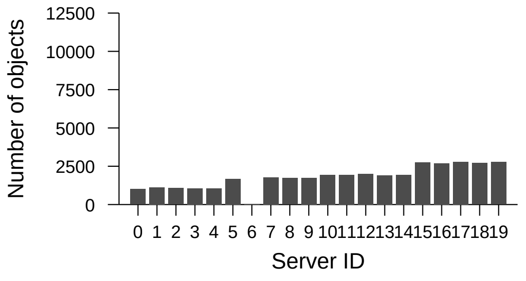


(b) Time per lookup with no weighting and exponential weighting

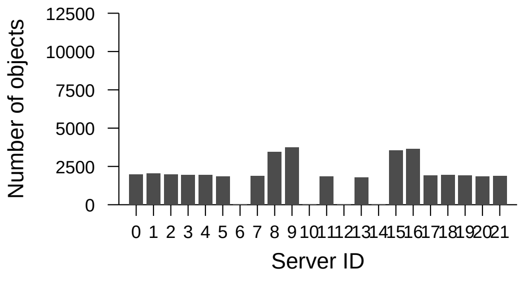
Figure 5. Time for looking up an object versus the number of server clusters in the system. All times computed on an Intel Pentium III 450.



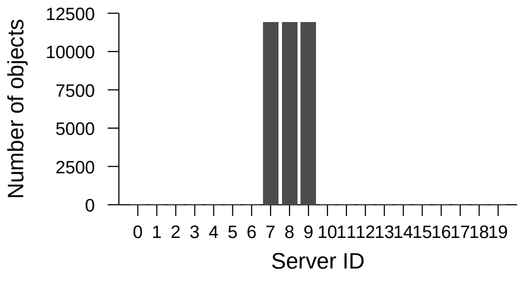
(a) Server 6 fails in a system with 4 evenly weighted clusters of 5 servers



(b) Server 6 fails in a system with 4 clusters of 5 servers, each cluster having increasing weight



(c) Server 6 fails in a system with 2 clusters of 5 servers, and 1 cluster of 12 servers. The failed server is in the the cluster of 12 servers.



(d) Server 6 fails in a system with 4 clusters of 5 servers, where object replicas are distributed to adjacent servers

Figure 6. The distribution of the replicas of objects stored on a failed server, where the server fails under different system configurations. A total of 300,000 objects are stored in the system.

objects on server 6. This occurs because the cluster with which server 6 was added is of size 12, which is a composite number ($3 \times 2^2 = 12$). Depending on the degree of replication and the number of distinct prime factors of the size of the cluster, if the size of a cluster is composite, some “empty spots” may occur in the cluster.

Even in when the number is a composite number, the objects are distributed relatively uniformly over most of the servers. Clearly such a distribution is far superior to a simplistic sequential distribution as illustrated in Figure 6(d), in which a few servers in the system ($R-1$ where R is the degree of replication, to be exact) will take on all of the load from the failed server. Instead, our algorithm distributes load from failed servers nearly uniformly over all of the working servers in the system.

6 Operational Issues

Our algorithm easily supports two desirable features for large-scale storage systems: online reconfiguration for load balancing, and variable degrees of replication for different objects.

6.1 Online Reconfiguration

Our algorithm easily allows load balancing to be done online while the system is still servicing object requests. The basic mechanism is to identify all of the "sets" that will move from an existing disk to a new one; this can be done by iterating over all possible values of x to identify those sets that will move. Note that our balancing algorithm will never move any objects from one existing disk to another existing disk; objects are only moved to new disks. This identification pass is very quick, particularly when compared to the time required to actually copy objects from one disk to another. During the process of adding disks, there are two basic reasons why the client might not locate the object at the correct server.

First, server clusters may have been reconfigured, but the client may not have updated its algorithm configuration and server map. In that case, the client can receive an updated configuration from the server from which it requested the object in question, and then re-run our algorithm using the new configuration.

Second, the client may have the most recent configuration, but the desired object has not yet been moved to the correct server. In that case, if the client thought that the object replica should be located in cluster j, but did not find it, it can simply continue searching as if cluster j had not been added yet. Once it finds the object, it can write the object in the correct location and delete it from the old one.

Different semantics for object locking and configuration locking will be necessary depending on other parameters in the system, such as the commit protocol used, but our algorithm is equally suited for online or batch reorganization.

6.2 Adjustable Replication

Our algorithm allows the degree of replication of any or all of the objects to vary over time with the following constraint—when the system is initially configured, the administrator must set the maximum degree of replication. This value can be no more than the size of the initial cluster (since we must have a unique location in which to place all replicas). The client can then decide on a per object basis how many replicas to place. If it places fewer than the maximum number possible, the spots for the remaining replicas can be used if a higher degree of replication is desired at a later time. Practically speaking, a client might use per-file metadata to determine the degree of replication of the different objects which compose a file in an OBSD.

7 Future Work

Our algorithm distributes data evenly and handles disk failures well, but there are further issues we are currently investigating. We are studying a more efficient parameterizable random number generation or hashing function, which will make the worst case performance of the algorithm $O(n)$. In addition, we are studying a modification to the algorithm which will allow for cluster removal. In exchange for this capability, the algorithm will need to look up all R replicas at once. This should not significantly affect performance if locations are cached after they are calculated.

We are also considering the exact protocols for the distribution of new cluster configuration information. These protocols will not require any global locks on clients, and in some cases where optimistic locking semantics are acceptable, will not require any locks at all.

We are considering different read/write semantics for different types of storage systems, and are integrating this algorithm into a massively scalable cluster file system.

Finally, we are considering a fast-recovery technique that automatically creates an extra replica of any object affected by a failure in order to significantly increase the mean time to failure for a given degree of replication [27].

8 Conclusions

The algorithm described in this paper exhibits excellent performance and distributes data in a highly reliable way. It also provides for optimal utilization of storage with increasing storage capacity, and achieves balanced distribution by moving as little data as possible. The use of weighting allows systems to be built from heterogeneous clusters of servers. In addition, by using replica identifiers to indicate the location of different stripes of an object, we can also use our algorithm to place stripes for Reed-Solomon coding or other similar striping and data protection schemes. Using these techniques, it will be possible to build multi-petabyte storage systems that can grow in capacity and overall performance over time while balancing load over both old and new components.

Acknowledgments

Ethan Miller was supported in part by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714.

References

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[2] Y. Breitbart, R. Vingralek, and G. Weikum. Load control in scalable distributed file structures. *Distributed and Parallel Databases*, 4(4):319–354, 1996.

[3] A. Brinkmann, K. Salzwedel, and C. Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–128. ACM Press, 2000. Extended Abstract.

[4] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, adaptive placement schemes for non-uniform capacities. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 53–62, Winnipeg, Manitoba, Canada, Aug. 2002.

[5] S.-C. Chau and A. W.-C. Fu. A gracefully degradable declustered RAID architecture. *Cluster Computing Journal*, 5(1):97–105, 2002.

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), June 1994.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[8] P. D. Coddington. Random number generators for parallel computers. *NHSE Review*, 1(2), 1996.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, Massachusetts, 2001.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, Banff, Canada, Oct. 2001. ACM.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, Oct. 1998.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture—A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. Wes. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[14] B. Kröll and P. Widmayer. Distributing a search tree among a growing number of processors. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 265–276. ACM Press, 1994.

[15] B. Kröll and P. Widmayer. Balanced distributed search trees do not exist. In *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, pages 50–61. Springer, Aug. 1995.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Nov. 2000. ACM.

[17] W. Litwin, J. Menon, and T. Risch. LH* schemes with scalable availability. Technical Report RJ 10121 (91937), IBM Research, Almaden Center, May 1998.

[18] W. Litwin, M. Neimat, G. Levy, S. Ndiaye, T. Seck, and T. Schwarz. LH*$_{\mathrm{S}}$: a high-availability and high-security scalable distributed data structure. In *Proceedings of the 7th International Workshop on Research Issues in Data Engineering, 1997*, pages 141–150, Birmingham, UK, Apr. 1997. IEEE.

[19] W. Litwin and M.-A. Neimat. High-availability LH* schemes with mirroring. In *Proceedings of the Conference on Cooperative Information Systems*, pages 196–205, 1996.

[20] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH*—a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[21] W. Litwin and T. Risch. LH*g: a high-availability scalable distributed data structure by record grouping. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):923–927, 2002.

[22] W. Litwin and T. Schwarz. LH*$_{\mathrm{RS}}$: A high-availability scalable distributed data structure using Reed Solomon codes. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 237–248, Dallas, TX, May 2000. ACM.

[23] M. Ripeanu, A. Iamnitchi, and I. Foster. Mapping the Gnutella network. *IEEE Internet Computing*, 6(1):50–57, Aug. 2002.

[24] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 188–201, Banff, Canada, Oct. 2001. ACM.

[25] R. O. Weber. Information technology—SCSI object-based storage device commands (OSD). Technical Council Proposal Document T10/1355-D, Technical Committee T10, Aug. 2002.

[26] B. A. Wichmann and I. D. Hill. Algorithm AS 183: An efficient and portable pseudo-random number generator. *Applied Statistics*, 31(2):188–190, 1982.

[27] Q. Xin, E. L. Miller, D. D. E. Long, S. A. Brandt, T. Schwarz, and W. Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*. IEEE, Apr. 2003. To appear.