

RUSH: Balanced, Decentralized Distribution for Replicated Data in Scalable Storage Clusters

R. J. Honicky
honicky@cs.ucsc.edu

Ethan L. Miller
elm@cs.ucsc.edu

Storage Systems Research Center
Jack Baskin School of Engineering
University of California, Santa Cruz

Abstract

Typical algorithms for decentralized data distribution work best in a system that is fully built before it first used; adding or removing components results in either extensive reorganization of data or load imbalance in the system. We have developed a decentralized algorithm, **RUSH** (Replication Under Scalable Hashing), that maps replicated objects to a scalable collection of storage servers or disks. RUSH distributes objects to servers evenly, redistributing as few objects as possible when new servers are added or existing servers are removed to preserve this balanced distribution. It guarantees that replicas of a particular object are not placed on the same server, and allows servers to have different “weights,” distributing more objects to servers with higher weights. The algorithm is very fast, and scales with the number of server groups added to the system. Because there is no central directory, clients can compute data locations in parallel, allowing thousands of clients to access objects on thousands of servers simultaneously.

1 Introduction

As the use of large distributed systems and large-scale clusters of commodity computers has increased, significant research has been devoted toward designing scalable distributed storage systems. However, 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.

Scalable distributed data structures (SDDSs) are partic-

ularly important for modern high-performance computing because of the development of computing and storage clusters built from thousands of individual nodes. In such systems, the use of a central directory for locating data would result in a severe bottleneck, reducing overall performance. As discussed in Section 7, existing SDDSs share three crucial properties that allow them to provide good performance in such large, scalable environments:

1. A file expands to new servers gracefully, and only when servers already used are efficiently loaded.
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 because a file that expands to new servers based on storage demands rather than on resource availability presents a 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. rather than waiting for the system to decide to take advantage of the new resources based on algorithmic characteristics and parameters that they do not understand.

This paper describes a decentralized algorithm, **RUSH** (Replication Under Scalable Hashing), that maps replicated objects to a scalable collection of storage servers or disks. RUSH distributes objects to disks evenly, redistributing as few objects as possible when new disks are added or existing disks are removed to preserve this even distribution. It guarantees that replicas of a particular object are not placed on the same server, and allows for

servers to have different “weights,” distributing more objects to servers with higher weights. The algorithm is very fast, and scales with the number of disk groups added to the system. Because there is no central directory, clients can compute data locations in parallel, allowing thousands of clients to access thousands of servers simultaneously.

RUSH builds on an algorithm we previously developed [13], adding the ability to remove disk clusters and improving the evenness of data distribution. The ability to remove disk clusters is crucial for long-lived storage systems because it allows system designers to build evolving systems, adding new capacity, and retire aging servers as necessary while preserving balanced data distribution and replication. Like our previous algorithm, RUSH allows objects to have an arbitrary, adjustable degree of replication; RUSH merely provides a mapping, allowing the storage system to decide how many replicas should actually be instantiated. Because replicas are distributed evenly to all of the disks in the system, 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.

2 Object-based Storage Systems

NASD¹-based storage systems [11, 12] 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 (or larger) 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 often called *object-based storage devices* (OSDs). We assume that the storage system on which our algorithm runs is built from OSDs.

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 OSD and the object returned to the client. By

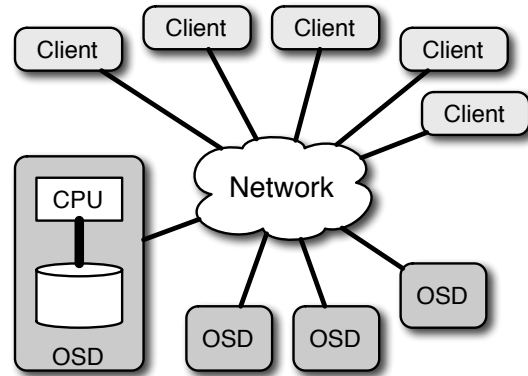


Figure 1: A typical NASD-based storage system

restricting the magnitude of x to a relatively small number, typically in the range 10^6 – 10^7 , we make the object balancing described in Section 6.1 simpler to implement without losing the desirable balancing characteristics of the algorithm.

Most previous work has either assumed that storage is static, or that storage is added for additional capacity. We believe that disks will be added and removed for several reasons, including better performance and system aging as well as capacity, requiring that objects be redistributed. If objects are not rebalanced when the disks are 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. Clearly, objects must be redistributed when storage is removed to prevent the loss of data.

3 Object Placement Algorithms

The major goals of an object placement algorithm are to provide a decentralized algorithm that preserves load balance on a collection of storage servers in the face of changes in the configuration of the storage system. Our object placement algorithms work on a set of *servers*, each of which is a storage system capable of servicing client requests for objects; in our model, servers correspond to OSD disks, but this is not a requirement. Servers are partitioned into *sub-clusters*; sub-clusters consist of identical servers that are added, removed, and reweighted as a group. The entire storage system consists of multiple server sub-clusters, accreted over time.

We assume that very large storage systems that use RUSH will be built over time. After adding or removing a sub-cluster or changing the weight of a sub-cluster, the system must *reorganize*, relocating some objects from their current servers to their new servers. This process may take some time; during this time, the system may be unbalanced, though it can continue to operate.

¹Network Attached Storage Device

c = the most recently added cluster while (objects remain to be mapped) for key k , decide how many object replicas belong in cluster c map each object which belongs in cluster c to some server in c change c to the cluster added immediately before c
--

Figure 2: Pseudo-code for mapping the replicas of an object to a server.

During reorganization, we would ideally like the system to move the minimum number of objects possible, in order to conserve network resources or minimize the amount of time offline in the event of an offline reorganization. A reorganization is considered *optimal* if moving fewer objects would result in a system that is not balanced. Intuitively, this corresponds to the situation in which the system moves sufficient objects to or from the sub-cluster whose weight was changed so that the system is once again balanced, assuming that objects only move between the affected sub-cluster and the rest of the system.

3.1 Algorithm Fundamentals

RUSH is based upon a simple principle: every time a new sub-cluster is added or the weight of an existing cluster is changed, use a parameterized hash function to decide which objects must move to or from the sub-cluster to maintain balance in the system. Once objects have been identified, use another parameterized hash function to decide on the destination for the objects within a particular cluster.

Object lookup and placement uses this process in reverse. For a given object, the algorithm asks “Would replicas of this object have moved to the most recently added sub-cluster?” If so, the algorithm uses a hash function to locate the object replicas within the sub-cluster. If the replicas are not in this cluster, they must be in the portion of the system not yet examined, so the process is repeated for the next newest sub-cluster. The basic algorithm, shown in Figure 2, may seem slow because it must iterate over all clusters, but, as demonstrated in Section 5.1, it is actually fast in practice.

The algorithm in Figure 2 would be intractable if it had to be computed for each object in the system, as would be needed to identify all of the objects on a particular server. Object IDs are therefore mapped into a fixed number of *keys*, as described in Section 2. Our distribution algorithm is then run over each groups, where a group is defined as all objects in the system that share the same key. It is for this reason that we typically limit key range to 10^6 – 10^7 ; by treating all objects with the same key as a unit, our algorithm can run quickly.

R	Maximum number of replicas of a given object
c	Number of sub-clusters in the system
m_i	Number of servers in sub-cluster i
w_i	Weight assigned to a single server in sub-cluster i
$n = \sum_{i=0}^{c-1} m_i$	Number of servers in the system
$S = \{s_0, \dots, s_{n-1}\}$	Set of all server IDs in the system

Table 1: Parameters used to define data distribution in RUSH.

3.2 Cluster Weighting and Replication

In addition to allowing fast lookup and fast online reorganization, an algorithm designed to distribute data on large distributed storage systems must allow servers to be weighted differently, since we can expect that when new servers are required, the capacity or throughput of the servers will be different from that of existing servers. In order to take advantage of this increased capacity, we must be able to allocate more objects to more powerful servers.

Additionally, a good lookup algorithm should allow servers to be removed from the system once the cost of maintaining them becomes higher than the cost of replacing them. We achieve this by allowing something stronger: arbitrary reweighting of sub-clusters. Although a sub-cluster can be removed by setting its weight to zero, its weight could also be adjusted to account for configuration errors or other abnormal situations.

Finally, Xin, *et al.* report that the mean time to failure (of a single disk) in a petabyte-scale (10^{15} bytes) storage system will be approximately one day [27]. In order to prevent data loss, we must allow for data replication. Furthermore, the data replication scheme should guarantee that replicas of the same object get placed on different servers, or the effect of replication will be nullified.

4 Replication Under Scalable Hashing

To address all of these issues in scalable replicated hashing, we developed RUSH (Replication Under Scalable Hashing), an algorithm that allows cluster weighting and reweighting and replication in addition to supporting cluster reorganization and fast distributed lookup. The state of the entire storage system at a particular time can be defined by the parameters in Table 1.

4.1 The RUSH Algorithm

RUSH was designed specifically to address the problem of sub-cluster removal. In a system that grows over time, the newer servers will eventually contain most of the objects in the system, since disk capacity and throughput is increasing exponentially. The older disks, which are slower and smaller, will begin to fail as they age; eventually, the

cost of maintaining them will be higher than the benefit of keeping them. At that time, they should be retired.

As mentioned above, RUSH allows disks to be retired by solving a slightly more general problem: how to reweight a sub-cluster arbitrarily. In other words, RUSH allows the administrator of a system to change the weighting parameters on one or more sub-clusters. If one can reweight a sub-cluster, a sub-cluster’s weight can be set to 0, causing all objects in that sub-cluster to move to another sub-cluster and allowing the servers in that sub-cluster to be removed from the system. This weight adjustment, however, may come at the cost of moving slightly greater than the minimal number of objects necessary to keep the system balanced, depending on the parameters of the reorganization.

Recalling that R represents the number of replicas of some object, x is the object’s key, and S is the set of possible server ids, RUSH is a function that maps from a key x and some number of replicas R to an R -tuple of server IDs $\langle s_1, \dots, s_R \rangle$. RUSH follows the basic structure described in Figure 2. It examines sub-clusters in the system starting with the most recently added sub-cluster, and determines the number of replicas which belong in that sub-cluster based on a draw from the hypergeometric distribution, as described in Section 4.2. Once it has determined that a particular sub-cluster should contain u replicas of an object, it selects u servers randomly from that sub-cluster. If any of the replicas remain to be allocated (*i. e.*, $R - u > 0$), the algorithm continues with the next most recently added sub-cluster. Figure 3(a) shows the pseudo-code for the algorithm, and Figure 3(b) shows the pseudo-code for the selection algorithm.

The need to guarantee that R replicas get allocated for any value of x and R , provided that $R \leq n$, contributes to RUSH’s complexity. In order to make this guarantee, the algorithm computes a minimum number of servers from the current sub-cluster to be allocated, denoted as t . The parameters for the draws from the weighted hypergeometric distribution are adjusted by t , and t is added to the result of the draw. Weighting introduces a second complexity: instead of simply dealing with numbers of servers, we must keep track of cumulative weight and per-cluster weight for each cluster.

RUSH relies upon a subroutine which can choose k elements from a set of n integers without replacement. The algorithm, described by the pseudo-code in Figure 3(b), does this with time-complexity $\Theta(k)$, not including initialization which is $\Theta(n)$. The algorithm avoids paying the initialization cost more than once by simply resetting the values in the array to their original values once the array of chosen values has been constructed.

For `choose` to work correctly, it must choose k distinct numbers from the set $\{0 \dots n - 1\}$, giving each number in the set equal probability of being chosen. Infor-

mally, since k swaps a random element in the set with the last element in the array (set) and removes the last remaining element in the array (by decrementing the end-of-array index), `choose` cannot pick the same element twice. Furthermore, since in each iteration `choose` selects a random element from the array of $n - i$ elements (by generating a random integer in the range $0 \dots n - i - 1$ using a uniform random number generator), each of the remaining elements has an equal probability of being selected. The time complexity of `choose` can be seen by simply observing that there is one `for` loops in each of `choose` and `reset`, each of which executes k times and includes only elementary operations. As a result, choosing individual servers on which each replica of an object is placed has time complexity $\Theta(k)$.

Another crucial property upon which RUSH relies is that `choose` must produce the same sequence of numbers regardless of the number actually required. In other words, `choose` can be thought of as a function that produces an ordered list of servers for given values of key x and cluster j . This means that if we call `choose` with a different k parameters k_1 and k_2 , $k_1 > k_2$, the first k_2 “choices” will be the same for both calls—the k_2 -prefix of the results of the two calls will be identical. This property is vital for ensuring that a minimal number of objects are moved when the weight of a cluster is changed.

RUSH requires, in the worst case, c iterations through its main loop to identify all of the sub-clusters that contain replicas of a particular object. For each sub-cluster, it must perform an iterative hypergeometric draw at a cost of $O(R)$, and it must identify the servers within the sub-cluster that contain the object, requiring total time $O(R)$, for a worst-case cost of $O(Rc)$. In reality, however, the cost is usually significantly less than $O(Rc)$ because some replicas are found during the first iterations, reducing the number of iterations the hypergeometric draw must run for later clusters, as demonstrated in Section 5.1.

4.2 Drawing from the Weighted Hypergeometric Distribution

The hypergeometric distribution is a discrete distribution which is similar to the binomial distribution, except that samples are taken *without replacement*. While for large populations, sampling with or without replacement makes little difference, in small populations—less than a thousand values—replacement makes a marked difference. Since our algorithm will typically need to sample from small populations when deciding whether or not to place an object in a cluster, the hypergeometric distribution provides the most accurate samples.

The hypergeometric distribution function $H(r, n, N)$ is typically defined in terms of three parameters: r , the number of draws to make; n , the number of positives in the

```

def init()
  for  $j = 0$  to  $c - 1$ :
     $m'_j \leftarrow m_j w_j$ 
     $n'_j \leftarrow \sum_{i=0}^{j-1} m'_i$ 
  end for

  def getServers(  $x, R$  )
    for  $j = c - 1$  downto  $0$ 
      seed the random number generator with  $hash_1(x, j)$ 
       $t \leftarrow \max(0, R - n_j)$ 
      //  $H$  is a draw from the weighted hypergeometric distribution
       $u \leftarrow t + H(R - t, n'_j - t, m'_j + n'_j - t, w_j)$ 
      if  $u > 0$  then
        seed the random number generator with  $hash_2(x, j)$ 
         $y \leftarrow \text{choose}(u, m_j)$ 
        reset()
        add  $n_j$  to each element in  $y$ 
        append  $y$  to a list of chosen servers  $l$ 
         $R \leftarrow R - u$ 
      end if
    end for
  return  $l$ 

```

(a) An algorithm for mapping all replicas of an object to corresponding servers.

```

def initChoose(  $n$  )
   $a_{0..n-1} \leftarrow \{0, \dots, n - 1\}$ 

  def choose(  $k, n$  )
    for  $i = 0$  to  $k - 1$ 
      generate a random integer  $0 \leq z < (n - i)$ 
      // swap  $a_z$  and  $a_{n-i-1}$ 
       $r_i \leftarrow a_z$ 
       $a_z \leftarrow a_{n-i-1}$ 
       $a_{n-i-1} \leftarrow r_i$ 
    end for
  return  $r$ 

  def reset
    for  $i \in \{0, \dots, k - 1\}$ 
       $c \leftarrow a_{n-k+i}$ 
      if  $c < n - k$ :
         $a_c \leftarrow c$ 
         $a_{n-k+i} \leftarrow n - k + i$ 
    end for

```

(b) Algorithm for choosing k integers out of $\{0, \dots, n - 1\}$ without replacement.

Figure 3: Pseudo-code for the RUSH algorithm. Details on the hypergeometric draw are in Section 4.2.

pool before sampling begins; and N , the total number of positives and negatives in the pool, where $N \geq r$ and $N \geq n$. In the case of an unweighted system of disks, r corresponds to the number of replicas still needing to be placed, n corresponds to the number of disks in the sub-cluster being considered, and N corresponds to the number of disks in all of the sub-clusters added previous to the sub-cluster being considered, plus the number of disks in the sub-cluster being considered.

Intuitively, we must sample without replacement because once we have decided to place an object replica on a particular server, that server is no longer available for placement of other replicas of the same object. In other words, if we sample a “positive” then we have chosen a server in the sub-cluster being considered to hold one of the object replicas (although we have not determined which server in the sub-cluster will hold the object). Alternatively, if we sample a “negative,” then we have chosen a server for the object replica in some sub-cluster which was added to the system before the sub-cluster being considered.

There are many ways to sample from the hypergeometric distribution, including iterative sampling, ICDF sampling, and rejection sampling [25]. RUSH, however, also requires hypergeometric draws which are both monotonic with respect to changes in the number of samples taken (r), and which support weighted draws, where the weights can be any real number. These requirements force the use of the iterative sampling technique; fortunately, the pa-

rameters to the distribution function that RUSH uses mean that iterative sampling typically performs better than the other techniques.

The first requirement, that the sampling algorithm must be monotonic with respect to the changes in the number of samples taken, means that first of all, if we increase the r parameter by some value δ , we are guaranteed that $H(r + \delta, n, N) \geq H(r, n, N)$, and secondly if we increase N by some value Δ , we are guaranteed that $H(r, n, N) \geq H(r, n, N + \Delta)$. Monotonicity is important because we need to ensure that as the weight of a particular sub-cluster decreases, reducing the ratio $\frac{n}{N}$, the number of replicas of a particular object which get stored in that sub-cluster either decreases or stays the same. If this were not true, some objects might move between existing clusters, rather than only moving to a newly added sub-cluster. In particular, if $H(r - \delta, n, N) > H(r, n, N)$, an existing cluster whose relative weight decreases might nonetheless have additional replicas allocated to it, resulting in replica movement between existing clusters.

Clearly, our sampling algorithm must support weighting because we can arbitrarily weight sub-clusters. The number of “positives” in a cluster must represent the total weight possessed by a particular sub-cluster, rather than simply representing the number of servers. If there are n servers in a sub-cluster with weight w , the subcluster has total weight $n \cdot w$. Each time a successful draw is made from the pool of N servers, the total weight of the servers still available for placing other replicas decreases by w

```

def draw ( $r, n, N, w$ )
   $c \leftarrow 0$ 
  for  $i = 0$  to  $r - 1$ 
    generate a random number  $z \in [0, 1]$ 
    if  $z \leq \frac{n}{N}$  then
       $c \leftarrow c + 1$ 
       $n \leftarrow n - w$ 
    end if
     $N \leftarrow N - w$ 
  return  $c$ 

```

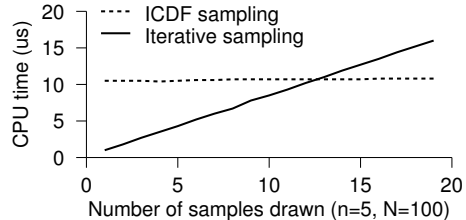
Figure 4: An iterative algorithm for sampling from the weighted hypergeometric distribution.

rather than 1, making the new total weight $(n - 1)w$. Both of these requirements combined mean that iterative sampling is the only technique that RUSH can use for its hypergeometric distribution.

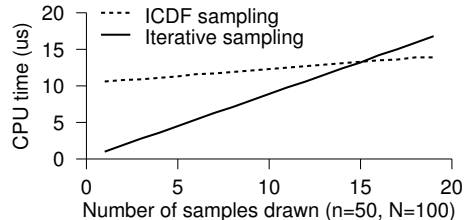
Iterative sampling is the most intuitive way to sample from the hypergeometric distribution. Simply make r random draws from the set $[0, 1]$, and compare them with the ratio $\frac{n}{N}$, adjusting n and N after each draw according to whether draws are positives (i.e. the draw is less than or equal to $\frac{n}{N}$) or negatives (i.e. the draw is greater than $\frac{n}{N}$). Pseudo-code for an iterative algorithm for making weighted hypergeometric draws is shown in Figure 4. The algorithm supports weighting quite elegantly, and fits with our intuitive understanding of weighting. It is also monotonic, assuming that we use the same random number sequence for the draws of z each time. Since we seed the random number generator with the same value regardless of weighting, the same number sequence will be generated.

The monotonicity of the sampling algorithm in Figure 4 (`draw`) with respect to changes in the ratio $\frac{n}{N}$ is critical. Recall that for the sampling algorithm to monotone it must be the case that if we increase the r parameter by some value δ , we are guaranteed that $H(r + \delta, n, N) \geq H(r, n, N)$, and if we increase N by some value Δ , we are guaranteed that $H(r, n, N) \geq H(r, n, N + \Delta)$. We omit the proof of the monotonicity of `draw` in the interest of brevity, but intuitively we can see that if r goes up, then we will sample at least as many positives, and if N goes up, then the ratio $\frac{n}{N}$ goes down, so we should sample fewer positives.

The only potential issue with the `draw` is that it can be slow if the degree of replication r is large. As Figure 5 shows, however, the iterative sampling method is actually *faster* than ICDF sampling as long as $r < 12$ because of the overhead associated with other technique. Furthermore, depending on the parameters, the crossover point may be even greater. For typical values of r such as 3–5, the iterative method is significantly faster.



(a) Computation time for ICDF sampling and iterative sampling with $n = 5$ and $N = 100$.



(b) Computation time for ICDF sampling and iterative sampling with $n = 50$ and $N = 100$.

Figure 5: CPU time required for ICDF and iterative sampling for different parameters. Even though ICDF uses a single “sample,” Iterative sampling is faster unless r is at least 12 or more, depending on the parameters.

4.3 Analysis of Data Distribution in RUSH

To show that RUSH allocates object replicas according to the distribution of weights over all of the servers in the system when a new sub-cluster is added to the system, we outline a simple inductive proof.

In the base case, there is one sub-cluster in the system. If there is one sub cluster in the system, the all of the servers in the system must, by definition, have the same weight. Since all of the servers have the same weight, the distribution of object replicas over the each server should be uniform. But since the `choose` function in Figure 3(b) distributes objects uniformly over a sub-cluster (see Section 4.1), and there is only a single sub-cluster, the objects must be distributed according to the weights assigned to each server, which are assigned uniformly across a cluster.

If sub-cluster c is added to a system that is already balanced, we must now ensure that the new distribution of objects will remain balanced. Initially, the total weight of all of the existing servers is $W = \sum_{j=0}^{c-1} w_j m_j$, and the total weight of the servers in the new sub-cluster is $W_c = w_c m_c$.

When drawing from the weighted hypergeometric distribution described in Section 4.2, each positive draw reduces the total weight of all of the positives by w , and each draw (either positive or negative) reduces the total weight by w . Similarly, when selecting where to place an object replica, each time we choose a server in sub-cluster c , we reduce the number of servers available for placing replicas. In that case, the effective value of W will decrease by w_c . Each time we select a server,

the value of $W + W_c$ decrease by w_c . Thus if we make a draw $X \sim H(R, W_c, W + W_c, w_c)$, the expectation of X is

$$\begin{aligned} E(X) &= \sum_{k=0}^R kh(k; R, W_c, W + W_c, w_c) \\ &= \sum_{k=0}^R k \frac{W_c! R! (W + W_c - W_c)! (W + W_c - R)!}{(W + W_c)! k! (W_c - k)! (R - k)! (W + W_c - W_c - R + k)!} \\ &= R \cdot \frac{W}{W_c + W}. \end{aligned}$$

The expectation of X is therefore the ratio between the total weight of the sub-cluster (W_c), and the total weight of all of the previously added sub-clusters, plus the sub-cluster under consideration ($W + W_c$), multiplied by the number of replicas (R). Since this is the expected number of replicas that will be placed on sub-cluster c for a particular key x , the new sub-cluster should receive the number of object replicas necessary to keep the system balanced.

This does not automatically guarantee that the other servers remain balanced. For example, if all $R \cdot \frac{W}{W_c + W}$ replicas came from the same sub-cluster in a system with 100 sub-clusters, this would not leave the system balanced. Since objects allocated to the new sub-cluster are chosen randomly, the probability of an object moving from a given server to the new sub-cluster is proportional to the number of objects located on that server.

In the case in which the weights of existing servers are explicitly adjusted, we can use the same argument as above: we simply start with the sub-cluster which was added first, and add the clusters with the new weights one at a time using the same inductive procedure.

Note that this process is only for the sake of proving correctness, rather than for determining the computational complexity of the algorithm. As we discuss below, the actual algorithm is typically optimal in the number of object replicas moved, and very fast, as shown in Section 5.1.

The optimality of our algorithm depends on the monotonicity of our hypergeometric draw. When a sub-cluster is added or the weight of a sub-cluster increases, the relative weight of every other sub-cluster must decrease. As described in Section 4.2, however, this draw is monotonic, so a sub-cluster that has its weight reduced cannot hold more replicas for a given object key than it did before, though it could hold fewer. The only sub-cluster that *can* accept more replicas for that object key is the sub-cluster whose relative weight increased.

Furthermore, as described in Section 4.1, the `choose` algorithm guarantees that if the number of replicas in the sub-cluster increases from k to some $k' > k$, the servers that `choose` picks will include the first k servers that were originally used. This means that objects need not move between servers in a single sub-cluster.

Since objects can only move to or from the cluster whose weight has changed, and objects need not move within sub-clusters, the number of objects moved is ap-

proximately the minimum necessary to keep the system balanced.

The analogous argument does not quite apply when a sub-cluster's weight is decreased: when a sub-cluster's weight increases, the number of objects allocated to every other sub-cluster increases slightly. For each of the other sub-clusters with higher weight, there is no guarantee that the new objects for which it is responsible will come from the sub-cluster whose weight decreased.

It can be shown that the expected number of objects moved in such a reorganization is no more than twice the optimal number, but since the proof is rather complex, we do not have space to present it here.

5 Operating Characteristics

To measure the operating characteristics of RUSH, we implemented it in a simulated storage system and measured its behavior under both normal conditions and node failure. This allowed us to calculate the CPU time required to do mappings and verify that load was balanced both in normal operation and after nodes had failed.

5.1 Performance

In order to quantify the real world performance of RUSH, we simulated a storage system that started with 10 disks, and added 100 sub-clusters containing 10 disks each. Note that performance figures would have been nearly identical regardless of the number of disks per sub-cluster; we could have just as easily added 100 sub-clusters with 1,000 disks each, and gotten the same performance results because the time complexity of the algorithm depends only upon the number of sub-clusters, not the number of servers.

For each configuration of the system, we simulated the lookup of one million different objects, each with four replicas. We then divided the total execution time by four million to get the per-object, per-replica lookup time. Using this metric allows us to generalize to systems with different replication factors, since the performance is linear in the replication factor.

The performance of RUSH depends on the weighting configuration of the sub-clusters. In the worst case, the vast majority of the weight in the system is concentrated in the oldest sub-cluster. In that case, most object lookups will need to check every sub-cluster before checking the last sub-cluster to locate all of the replicas.

The common case, however, will be that as per-disk capacity increases, the weight of sub-clusters will increase. The rate at which it increases depends on the frequency at which sub-clusters are added, and the rate of capacity growth in disks.

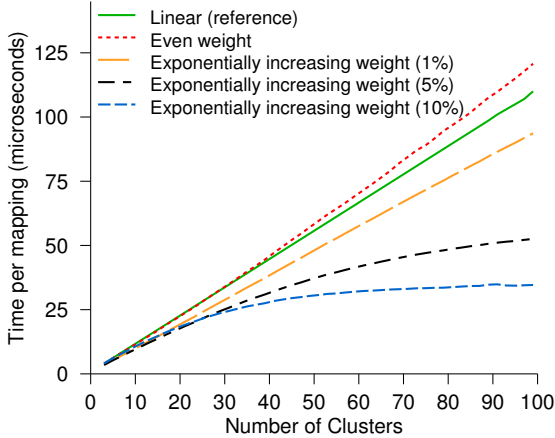


Figure 6: Time per lookup per replica as the number of clusters increases. The figure shows a linear function for reference.

Figure 6 shows the per-object per-replica lookup time for a system with 4 replicas per object, with various growth rates for the capacity of the most recently added cluster. In a system in which the weight of the most recently added cluster remains constant as clusters are added, performance degrades slightly super-linearly. Even a slight rate of increase in weight such as 1%, however, results in sub-linear degradation of performance. If the rate of increase is 10%, then performance is logarithmic in the number of clusters. Even with 100 clusters in the system, the amortized lookup time is less than $30\mu\text{s}$ on the slow 450 MHz Pentium III on which we ran these experiments.

5.2 Failure Resilience

When a server fails, other servers which contain replicas of the data held by the failed server must take responsibility for the the work done by the failed server. In very large clusters, the mean time between disk failures can be one day or less [27], so failure semantics of a data distribution algorithm are important even for day to day operations of a large disk cluster.

In a simplistic replication scheme where replicas of the objects for a particular server are simply distributed to (logically) nearby servers, the servers close to the failed disk will experience a severe spike in offered workload during a failure. A large spike in workload can in turn cause severe performance degradation, and possible aggravate further failures.

RUSH addresses this problem by distributing the replicas of the objects on a given server throughout the entire system, according to the distribution of weight in the system. When a server fails, the work performed by the failed server gets distributed throughout the entire system.

To verify that RUSH distributes object replicas for a

particular server throughout the system, we simulated a system with three sub-clusters. The first and third sub-clusters each have five disks, while the second one has twelve. These sizes were chosen to confirm that the evenness of data does not rely on the sub-cluster sizes. We tested systems both with even weight and with weight which increases with newly added clusters. We inserted 120,000 objects into the system and caused servers to fail, looking at the location of the object replicas which replicate data on the failed servers.

In Figure 7(a), we see the distribution of object replicas which replicate data from a single failed server. The graph show the distribution of replicas under two different configurations: even weight and increasing weight. When the weight is even, we can see that object replicas are distributed approximately evenly throughout non-failed servers, and when the weight increases, more object replicas are stored on the higher weight servers.

Figure 7(b) shows that the distribution of object replicas follows the distribution of weights even when multiple servers fail. Figure 7(c) illustrates that distributing replicas to nearby servers causes very uneven distribution of object replicas for a given server, and will cause a proportionately uneven workload in a failure situation. RUSH distributes data in such a way that when a server fails, the entire system shares workload from the failed disk.

5.3 Data Distribution

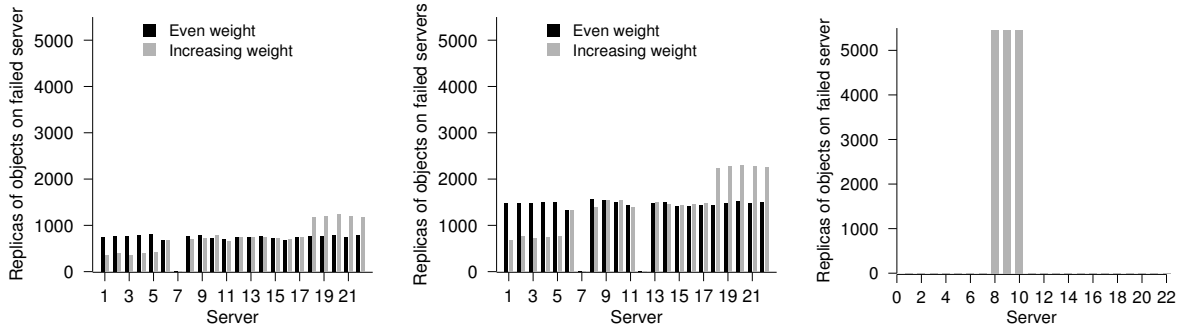
Although we have shown that the the expected number of objects in a particular server is proportional to the fraction of the total weight in the system owned by the server, we have not discussed how far the server will deviate from the expected value.

We can visually examine Figure 8 to confirm that as the number of servers increases, the objects indeed spread out to all of the servers. Furthermore, the even distribution of objects among servers in the same cluster suggests that the distribution of objects is indeed proportional to the fraction of the total weight owned by the server. To verify the accuracy of the distribution, however, we must introduce a metric of accuracy.

We measure the accuracy of the object distribution using the Root Mean Square Error (RMSE), a metric related to the standard deviation that measures the distance from the expected value rather than the mean value. It is normalized to reflect percentage difference instead of absolute difference to facilitate comparison between different expected values.

The Normalized Root Mean Square Error (NRMSE) is calculated as

$$E_c = \sqrt{\frac{\sum_{i=1}^{m_c} (y_c - \hat{y}_i)^2}{m_c}} \cdot \frac{1}{y_c},$$



(a) A single server (Server 7) has failed. Object distribution is shown for two system configurations: even weight and increasing weight. (b) Two servers (Server 7 and Server 12) have failed. Object distribution is shown for two system configurations: even weight and increasing weight. (c) A single server (Server 7) has failed. Object distribution is shown for a naive placement algorithm under which replicas are distributed to nearby servers.

Figure 7: The distribution of object replicas stored on failed servers. A total of 120,000 object replicas are stored in the system. There are three sub-clusters in the system, Servers 1–5, Servers 6–17 and Servers 18–22.

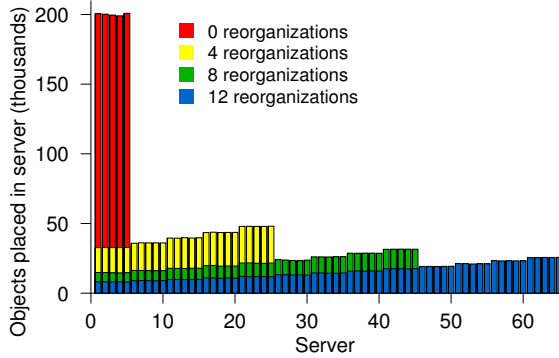


Figure 8: The distribution of one million objects for various numbers of sub-clusters, each containing five servers. The weight of the newer sub-clusters increases.

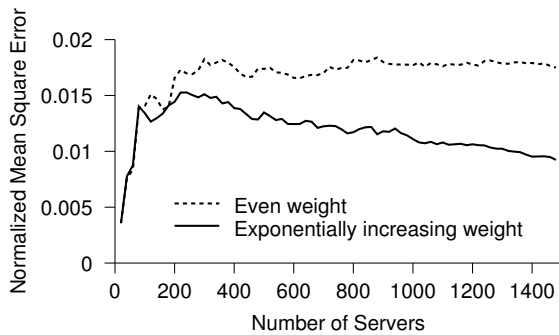


Figure 9: The average Normalized Root Mean Square Error as the number of servers and number of objects increase.

where E_c represents the NRMSE for some cluster c , y_c represents the expected number of objects per server for cluster c , \hat{y}_i represents the actual number of objects for a given server i , and m_c is the number of servers in the cluster.

Figure 9 shows the NRMSE averaged over all sub-clusters in a given system as the number of servers and the number of objects in the system increases. If the weight of each new sub-cluster remains constant, then the NRMSE remains relatively constant (after reaching a plateau). On the other hand, if the weight of each new sub-cluster increases exponentially (as we suspect it will in real systems), then the NRMSE decreases as the number of servers increases.

This graph suggests that we can trust the system to allocate objects according to the weights of each server, with an increasing degree of accuracy as the number of objects increases.

6 Operational Issues

6.1 Online Reconfiguration

RUSH 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 between disks; this can be done by iterating over all possible key values, typically 10^6 – 10^7 as described in Section 3.1, to identify those sets that will move. This identification pass is fast, relative 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 rerun 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, the client can proceed as if the most recent reorganization had not occurred. For example, if the most recent reorganization was the addition of a sub-cluster, and if the client thought that the object replica should be located in sub-cluster j , but did not find it, it can simply rerun the search as if sub-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. The clients must maintain state for both the old and new system configurations, but this can be done very efficiently by simply keeping the original configuration and the change applied to it until the reorganization is complete.

6.2 Adjustable Replication

Although the parameters to RUSH include the maximum number of replicas, the replication factors can be determined on a per-object basis. Practically speaking, a client might use per-file metadata to determine the degree of replication of the different objects which compose a file in order to increase reliability for particularly important files or decrease storage requirements for less important files such as temporary files. RUSH provides excellent support for adjustable replication factors because replicas are distributed evenly throughout the system, in contrast to a system that might use certain sets of servers for higher-numbered replicas. Assuming that an object’s key has no correlation to its replication factor, replicas from objects with high and low degrees of actual replication will be evenly spread throughout the system.

Another application of adjustable per-object replication is to reduce the number of replicas necessary to achieve a given mean time to data loss. If a particular server crashes, we can increase the replication factor of all the objects which are stored on that server. That way, the window of vulnerability (the time during which the replication factor of some objects is decreased due to a server failure) is shortened to the time it takes to increase the replication factor for all of the objects on the server. Since that process can be completely decentralized [27], the window of vulnerability will be significantly shortened, even if the administrator responsible for fixing the crashed server is not able to repair it for a long time.

More general error-correction codes can be used in place of replication to save storage space at the expense of decreasing small write speed. Our previous algorithm [13]

supports the use of “replica” numbers to label elements in a stripe by explicitly computing *which* replica is stored on which particular server. RUSH does not currently support this use of replication because there is no way to distinguish replicas from one another; however, we expect that future versions of RUSH will do so.

7 Related Work

Several different scalable distributed data structures have been proposed previously. Litwin, *et al.* have developed many variations on Linear Hashing (LH*) [17, 18, 20, 21, 22] The LH* variants are limited in two ways: they must split buckets, and they have no provision for buckets with different weights. LH* splits 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, resulting in sub-optimal disk utilization [2] and a “hot spot” of disk and network activity between the splitting node and the recipient. Other data structures such as DDH [10] suffer from similar splitting issues. Our algorithm, on the other hand, almost 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. Unlike RUSH, the LH* variants provide no mechanism for weighting different disks to take advantage of disks with heterogeneous capacity or throughput, a crucial requirement for storage clusters which grow over time. Breitbart, *et al.* [2] discuss a distributed file organization which resolves the issues of disk utilization in LH*, but does not propose any solution for data replication.

Kröll and Widmayer [15] propose Distributed Random Trees (DRTs), a type of SDDS that are optimized for more complex queries such as range queries and closest match rather than simple primary key lookup. DRTs support server weighting but face difficulties similar to the LH* variants with reorganization. DRTs do not explicitly support replication, though metadata replication is discussed, and have worst-case performance linear in the number of disks in the cluster. Any tree-based SDDS, including DRT, has an average-case performance with a lower bound of $\Omega(\sqrt{m})$ [16], where m is the number of *objects* stored by the system. Our algorithm has performance which grows linearly with the number of *groups of disks* added; performance is independent of the number of objects and, if disks are added in large groups, will approach constant time. Litwin, *et al.* also discuss a B-Tree based SDDS family called RP* [19]. It also suffers from the fundamental limitations of SDDS’s, and is bounded by $\Omega(\sqrt{m})$ in the average case [16].

Choy, *et al.* [7] describe algorithms for perfectly distribution of data to disks that move an optimally low number of objects when disks are added. However, these algo-

gorithms do not support necessary features such as weighting of disks, removal of disks, and replication. 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. Again, however, this method does not allow for the placement of replicas.

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 [9], PAST [26], Gnutella [24], and FreeNet [8] 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 [23] 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. The inefficiencies that 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). However, these features introduce far too much overhead for a tightly coupled mass object storage system.

Distributed file systems such as AFS [14] 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.

8 Future Work

We are currently investigating variations on RUSH that will reduce the lookup time, reducing the upper bound on lookup time. A new version of the algorithm will also

combine the best features of the two existing algorithms by allowing both identification of particular replicas and arbitrary reweighting and removal of existing sub-clusters of servers. The combination of these two features will further improve the suitability of RUSH for large-scale storage clusters.

One issue for RUSH is the exact protocols for the distribution of new cluster configuration information, and their interaction with the data commit protocols. These protocols will not require any global locks on clients, but care must be taken to ensure that no race conditions exist in the system.

We are designing a petabyte-scale storage system (see Figure 1), and are planning to use RUSH to distribute data to OSDs. As discussed in Section 6.2, we are developing a fast-recovery technique that automatically creates an extra replica of objects affected by a failure in order to significantly increase the mean time to data-loss for a given degree of replication. We are also investigating the suitability of RUSH for distributing metadata to our metadata servers to ensure scalability and balanced load.

9 Conclusions

RUSH is a fast, decentralized algorithm for placing and locating replicated objects in a scalable, distributed storage system. Data is spread to servers proportional to weights assigned by the system builder, and lookups can be done in microseconds by any client of the system keeping minimal state. Balanced object distribution is maintained as servers are added or removed; reorganization to preserve balance requires minimal object movement.

The use of RUSH to distribute objects in a petabyte-scale storage system will allow the system to be built over time with minimal disruption to continued use. Systems using RUSH can incorporate new servers, retire old servers, and change the fraction of objects stored on a set of servers, using changing technology to meet changing demands on the system. RUSH also permits the high-level file system to use as much replication is needed for each object, allowing it to dynamically set the number of replicas kept for each piece of data. In summary, RUSH can provide fast, decentralized object lookup while providing balanced data placement in distributed server clusters.

Acknowledgements

The research in this paper was supported in part by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714. We also thank the students and faculty of the Storage Systems Research Center for their help in preparing this paper.

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] D. M. Choy, R. Fagin, and L. Stockmeyer. Efficiently extendible mappings for balanced data distribution. *Algorithmica*, 16:215–232, 1996.
- [8] 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.
- [9] 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.
- [10] R. Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, pages 101–114, 1993.
- [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] G. A. Gibson and R. Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.
- [13] R. J. Honicky and E. L. Miller. A fast algorithm for online placement and reorganization of replicated data. In *Proceedings of the 17th International Parallel & Distributed Processing Symposium*, Nice, France, Apr. 2003.
- [14] 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.
- [15] 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.
- [16] 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.
- [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*_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, M.-A. Neimat, and D. Schneider. RP*: A family of order-preserving scalable distributed data structures. In *Proceedings of the 20th Conference on Very Large Databases (VLDB)*, pages 342–353, Santiago, Chile, 1994.
- [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*_{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] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the 2003 Conference on File and Storage Technologies (FAST)*, pages 1–14, Mar. 2003.
- [24] M. Ripeanu, A. Iamnitchi, and I. Foster. Mapping the Gnutella network. *IEEE Internet Computing*, 6(1):50–57, Aug. 2002.
- [25] B. D. Ripley. *Stochastic Simulation*. J. Wiley, 1987.
- [26] 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.
- [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*, pages 146–156, Apr. 2003.