Simple, Exact Placement of Data in Containers

Thomas Schwarz, S.J.
Universidad Católica del Uruguay
Montevideo, Uruguay
tschwarz@calprov.org

Ignacio Corderí Darrell D.E. Long
University of California
Santa Cruz, CA
{icorderi, darrell}@cs.ucsc.edu

Jehan-François Pâris
University of Houston
Houston, TX
paris@cs.uh.edu

***Abstract*—Storage and other systems frequently need to distribute objects equally over several sites or devices. While this is simple for a static system, organizing the distribution when additional containers (for example, hard drives, or web content delivery sites) become available is difficult. We present here a very simple scheme based on the factorial number system that allows equal, dynamic distribution of mirrored or replicated objects.**

I. Introduction

Placing resources on servers is a perennial problem in Computer Science. It is often modeled as the placing of balls into bins. One example that arises is the allocation of replicated resources to web servers. Here, the problem is not only the efficient use of server storage space, but also quality of service guarantees and a fit to the geographical distribution of demand. Another important example is storage virtualization, where we want to give the impression of an on-line, practically unlimited storage space. A storage center will have many different disks, often of different capacity and possibly with rather different access times.

We propose a general solution to the specific problem of how to evenly distribute k replicas of N objects among n bins such that (a) each bin receives the same number of objects, (b) two replicas never share the same bin, and (c) equal distribution of reconstruction load, by which we mean that if a bin fails and is replaced by a replacement bin, we can regenerate the replicas of the objects on the failed bins from the other bins in such a way that each bin receives the same amount of load. Since we do not know *a priori* the objects we will store and we allow insertion and deletion of objects, these properties refer to expected values.

Our solution is based on the basic properties of the factorial number system. Moving from an ordinary (binary, decimal, or hexadecimal) number system to the factorial number system is as simple and as fast as changing among ordinary number systems. As we will see, the factorial number systems presents a very simple way to randomly distribute objects among a given number of bins. We later expand our solution – at the cost of additional calculations – to containers of arbitrary sizes.

A first application of our technique is the distribution of resources to cloud data centers. We rent the same amount of storage space in different data centers and may want to be able to acquire more storage space not only by extending the space we rent in the centers that we already use, but also by renting space on new centers. A second application is the storage of metadata in the distributed RAM of a multicomputer. Here we can assume that all machines have the amount of memory. A third example is extensible hashing, where we want the buckets to have the same size. In the first example, replication is critical for data durability; in all three, it is essential for preventing temporal bottlenecks due to the popularity of specific items.

The remainder is organized as follows: Section II reviews previous work on resource placement; Section III introduces the factorial number; Section IV introduces our technique and Section V expands it to the case of containers with arbitrary sizes; finally Section VI has our conclusions.

II. Related Work

Litwin *et al.* extended linear hashing to resource placement in a variable number of servers [10]. The goal of LH* and its variants is constant time access to objects in a distributed system, with little and possibly inaccurate knowledge of clients over the state of the dynamic system. LH* schemes place objects into buckets but are not concerned with achieving equitable distribution nor do they take the storage capacity of the system nodes into account. LH* splits a predetermined bucket whenever it detects a bucket overflow, but not necessarily the overflowing bucket. If the total number of nodes used is not a power of two, then there are two classes of buckets, with the nodes in one class having twice the expected number of buckets as the other.

Fagin [5] proposed extensible hashing with similar goals. The difference is that in extensible hashing, any overflowing hash bucket can split, whereas the distributed version of linear hashing splits buckets in a fixed order. Both can be helpful in devising placement schemes. Both implement Scalable Distributed Data Structures (SDDS) [9], which allow files to expand to new servers gracefully without central information that limit access primitives such as search and insertion to only update a single client.

Plaxton *et al.* [12] addressed the problem of dynamic placement of replicated objects in the context of the web. They propose a hashed-suffix routing structure. Karger *et al.* [7] propose consistent hashing to improve on Plaxton's result. While their scheme yields a faithful distribution, it does not allow for replica placement. Chen *et al.* [4] built on Plaxton's work to choose the number and placement of replicas while satisfying QoS requirements and server capacity constraints.

Supported in part by Grant CCF-1219163, by the Department of Energy under Award Number DE-FC02-10ER26017/DE-SC0005417 and by the industrial members of the Storage Systems Research Center.

In the context of storage virtualization, Brinkmann *et al.* [2], [3] presented several schemes that present an excellent compromise among various goals (see below), but that are not absolutely faithful, *i.e.* are not expected to distribute objects equally among containers. Schindelhauer similarly improves on consistent hashing allowing storage containers of different capacities [13]. Based on their target application, Brinkmann and colleagues propose this list for an ideal placement scheme:

1) **Faithfulness:** The expected number of recourses placed in a bin is between $\lfloor(1-\epsilon)d_i \cdot m\rfloor$ and $\lceil(1+\epsilon)d_i \cdot m\rceil$ for all containers i, where ϵ can be made arbitrarily small.
2) **Time Efficiency:** The scheme can calculate the position of a resource efficiently.
3) **Compactness**: The amount of information needed to calculate is small. In particular, it should only depend on the number of containers and on the number of resources in a logarithmic way.
4) **Adaptivity**: After a change in the number of containers, the number of resources, or the storage capacities, the distribution of the resources over the containers can quickly adapt to recover faithfulness. A measure of success in achieving this goal is *competitiveness*. A placement strategy is called *c-competitive* if at most c times the number of resources are moved than in an optimal adaptive and perfectly faithful strategy.
5) **Obliviousness**: The placement of resources into containers only depends on the resource identifiers and the number and sizes of containers, not on the history of the system.

Honicky and Miller [6] proposed RUSH to place objects in an object-based storage system. Their main insight is the nature of updates in a storage system, since new storage containers are added in clusters. In practice, even a highly dynamic system will not undergo a great number of additions. An improved version, CRUSH, removes issues that make RUSH an insufficient scheme in practice [14].

In comparison to previous work, we propose a simpler (and mathematically more elegant scheme) that is absolutely faithful, time efficient, very compact, and oblivious, but not always optimally adaptive unless we add or remove one container from the ensemble.

III. Factorial Representation

The factorial number system [8] is a system with the mixed radices 1!, 2!, 3!, 4!, ... We denote the expression of a number x in this system with x_f. If a number x is given in this system as $x_f = (x_1, x_2, x_3, \ldots, x_{r-1}, x_r)$ then the digit x_i with index i, $1 \leq i \leq r$, lies between 0 and i: $0 \leq x \leq i$.

$$x = \sum_{\nu=1}^{r} x_\nu \cdot \nu!$$

As induction shows,

$$\sum_{\nu=1}^{r} \nu \cdot \nu! = (r+1)! - 1$$

```
def facRep(n, k=2):
   if n == 0:
      return []
   else:
      return [n % k] + facRep(n // k, k + 1)
```

Fig. 1. Algorithm to return the factorial representation of a non-negative number as a list in little-endian (least significant digit first) order.

This relationship implies the uniqueness of the representation, as we now prove via indirect proof. Assume that we have to representations $L = \sum_{\nu=1}^{r} a_\nu \nu!$ and $R = \sum_{\nu=1}^{r} b_\nu \nu!$ of the same number $L = R$ and let μ be the highest index for which they differ. This means that $a_\nu = b_\nu$ for all $\nu > \mu$ and that $a_\mu < b_\mu$ (or $a_\mu > b_\mu$, in which case we switch the roles of L and R.) Then $R - L = (b_\mu - a_\mu)\mu! + \sum_{\nu=1}^{\mu-1}(b_\nu - a_\nu)\nu!$. According to the inequality, the second addend $\sum_{\nu=1}^{\mu-1}(b_\nu - a_\nu)\nu!$ has an absolute value strictly smaller than $\mu!$. However, $b_\mu - a_\mu$ is not zero, and as an integer at least one. Therefore, the first addend $(b_\mu - a_\mu)\mu!$ has absolute value at least $\mu!$. If both representations represent the same number, then of course the sum should be zero, which is impossible. This contradiction proves the uniqueness of representation.

The calculation of the factorial representation can be done most easily by successive divisions with remainder by 2, 3, ... As an example, we take 1000. The least significant digit of the factorial representation results from dividing 1000 by 2 with remainder, yielding $1000 = 2 \cdot 500 + 0$. Proceeding, we divide by 3, giving us $500 = 3 \cdot 166 + 2$. The next step is division by 4, giving us $166 = 4 \cdot 41 + 2$, then division by 5, giving us $40 = 5 \cdot 8 + 1$, by 6, giving us $8 = 6 \cdot 1 + 2$ and finally by 7, giving us $1 = 7 \cdot 0 + 1$. We assemble the digits in ascending order in an array $[0, 2, 2, 1, 2, 1]$. We verify that

$$0 \cdot 1! + 2 \cdot 2! + 2 \cdot 3! + 1 \cdot 4! + 2 \cdot 5! + 1 \cdot 6! = 1000.$$

As Figure 1 shows, computing the factorial representation of any positive integer uses the same iterative approaches the algorithms for computing decimal and hexadecimal representations of numbers. There are two differences: First, our divisor changes at each iteration step; second, our result starts with the least significant digit contrary to the custom for most other representations of numbers.

IV. Algorithm

We have to distribute k replicas among N containers. We treat first the case of a single replica ($k = 1$), since it is easier to understand. We then generalize to an arbitrary number of replicas.

A. Single Replica

We describe the placement using a system that starts with one container, bin 0, and adds successively other containers of equal size, bin 1, bin 2, ... The number of containers is the *state* of the system. The location of a resource is a function that only depends on the state and the Resource Identifier (RID)

```
def assign (RID, nrBin):
   retval = 0
   for i, digit in enumerate(facRep(RID)):
      if digit == 0:
         retval = i
      if i >= nrBin:
         break
   return retval
```

Fig. 2. Algorithm to calculate placement of a resource with `RID` and `nrBin` containers.

```
def assignK (RID, rep, nrBin, nrRep):
   retVal = rep
   for i, digit in enumerate(facRep(RID)):
      if digit == rep and i >= nrRep:
         retVal = i
      if i >= nrBin:
         break
   return retVal
```

Fig. 3. Algorithm to calculate placement of replicas. The parameters are `RID`, the identity of the resource, `rep`, the identity of the replica, `nrBin`, the number of containers, and `nrRep`, the total number of replica.

of the resource to be stored. We assume that RIDs behave like random numbers. For example, they could be SHA-1 or even MD5 hashes of the unique name of the resources.

In State 1, we have only one container and all resources are located in bin 0. If we add a second container, we enter State 2 and will have to move (about) half of the resources to bin 1 in order to rebalance the load. To decide which resources should be moved, we consult the first digit x_1 of the factorial representation of all RIDs. As we obtained this digit by computing the RID modulo 2, its sole possible values are zero and one. If the value is 0, then we move the resource to bin 1, otherwise we leave it in bin 0.

Adding a third container bring us to State 3. We need to move (about) one third each of the objects in bin 0 and bin 1 to bin 2. We select them based on the second digit x_2 of the factorial representation of their RID, which can only be equal to 0, 1, or 2. As long as RIDs behave as a random number (which would be the case if they were hashes of the resource names), each of these digits will be selected with equal probability and independently of the previous digit. If x_2 equals 0, then we move the resource to bin 2.

Should a fourth container become available, we would enter State 4. Since the third digit x_3 of the factorial representation of each RID can only be equal to 0, 1, 2, or 3, moving to new bin all objects whose RID is equal to 0, would move about one fourth of the contents of bins 1 to 3 into the bin 4.

In general, we assign an object with RID x to bin i, if the index of the *last* occurrence of 0 in the list representation $\langle x_0, x_1, x_2, x_3, \ldots \rangle$ is equal to i.

By induction, we conclude that the expected values of the number of resources in each container are equal. We also note that each addition of a container leads to the minimum number of movements necessary in order to balance the load, as we only move to the new container, but not among the old containers. When we lower the number of containers, we undo the previous extensions, as the assignment of resources to containers depends only on state and RID. We give the algorithm as pseudo-code in Figure 2.

B. Several Replicas

We now treat the case of k replicas. Each replica has a *replica number* r (`rep` in Figure 3) from 0 to $k-1$. We never want to store replicas of the same object in the same container and need therefore at least k containers to start. If we are in this situation, we assign the replica with replica number r to bin r. Clearly, in this initial state, all containers have the same number of things to store, namely one replica per object.

If an additional container becomes available, we need to either let the replicas of a certain object stay in their current location or move one of the replicas into the new bin. In order to balance the load of the containers, we need to move $1/(k+1)$ of the contents of each old container to the new container. Thus, for a given object, we should not move any of the replicas with probability $1/(k+1)$ and we should move exactly one replica, but not more, with probability $k/(k+1)$. In the later case, we need to pick the replica to be moved with equal probability. We use the digit x_k of the factorial representation of the RID of the object to make the decision. This digit corresponds to the radix $(k)!$ and has a value in $\{0, \ldots, k\}$. If this digit has a value $x_k = k$, then we do not move any replica. Otherwise, we move the replica with replica number x_k.

Now assume an additional container, bin $k+1$ becomes available. Again, we need to move an equal proportion, in this case $1/(k+2)$, of the contents of the old containers to the new container. We use the digit x_{k+1} of the factorial representation of the RID of a resource. If $x_{k+1} \geq k$, then we leave the replicas of the resource where they currently are. Otherwise, we move the replica with replica number x_{k+1} to the new container, bin $k+1$.

In general, when we introduce bin l, we consult the digit x_l in the factorial representation of the RID of the resource. If the digit is smaller than k, we move the replica with replica number x_l to the new bin, otherwise, we do not move any replicas. As at most one replica is moved to a certain container when that container becomes available, and as initially, replicas of the same object are in different containers, our algorithm never places two replicas of the same object into the same bin.

As an example, consider an object with RID 12345678910. Its factorial representation is $[0, 2, 3, 2, 1, 3, 3, 3, 1, 3, 9, 12, 1]$. (These digits form the upper row of numbers in Figure 4). We assume that we place three replicas, starting out with 3 containers. We place the replicas 0, 1, and 2 in bins 0, 1 and 2, respectively (Figure 4, upper row). When we add one container, bin 3, we use the fourth-least significant digit of

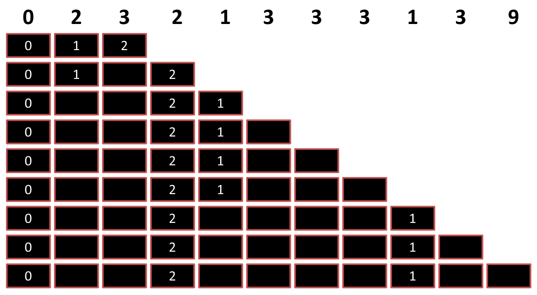

Fig. 4. Replica distribution for RID 12345678910 for three to eleven containers.

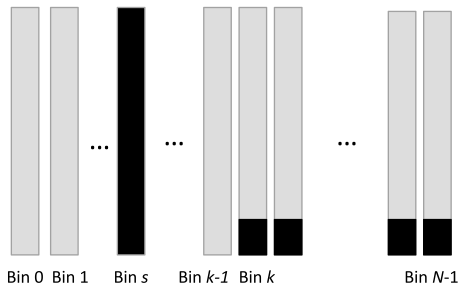

Fig. 5. Distribution of the s-th replica among N bins.

the factorial representation, namely 2. Since this is a replica number, we place replica 2 in the new bin. (Figure 4, second row). The next digit, 1 corresponds to introducing the fifth container. It leads us to place replica 1 into bin 4. The three digits 3 that follow mean that for this object, no replica changes container for the next three extensions. However, when we introduce bin 8, we have 1 as the corresponding digit, and we switch that replica to bin 8. The last row of Figure 4 shows the distribution of replicas with eleven containers.

The average number of objects in each container is equal, as we now argue. If we have k replicas and k containers, then each container contains the same number of objects. For an inductive step, we assume that our algorithm distributes the objects evenly over N containers and that we add a new container to the ensemble. If the new container is bin $N+1$, we see that it contains the first, second, third, etc. replica in equal proportions. In the general case, the new container will contain objects with RID identifier whose N^{th} factorial digit is in the set $\{0, 1, \ldots k-1\}$. This places a replica of $k/(N+1)$ of all objects into the new container. As there are k replica per container, $1/(N+1)$ of the contents of the system is placed in the new container. Furthermore, the proportion of first, second, third, etc. replica of objects is still the same. One of the first k containers (bins 0, 1, …, $k-1$) looses an object to the new container with probability $1/(N+1)$, moving $1/N \cdot 1/(N+1)$ of all contents, and reducing its share of total contents from $1/N$ to $1/(N+1)$. Another previous container (bins k, …, $N-1$) looses a first replica (with replica number 0) to the new container with probability $1/(N+1)$, and with the same probability a second, third, etc. replica. Consequentially, it looses $1/(N+1)$ of its contents, amounting to $1/N \cdot 1/(N+1)$ of total contents. We have shown by induction that *each container contains (in expectation) the same number of objects*. Additionally, we have shown that the contents of each added container (bins k, $k+1, \ldots$) contains replicas in the same proportion. For a specific replica with replica number s, it will be located with probability s/N in bin s and with probability $1/N$ in one of the bins k, $k+1$, …, $N-1$ (Figure 5).

The amount of data movement during an expansion from $N-1$ to N containers is expected to be $1/N$ of the contents of each old container to the new container. This is optimal.

Assume that the system requests contents by randomly selecting a replica number and then ask for the replica at the container storing it and assume now that a container is inaccessible. If our strategy selects replica r for a certain object and that copy is located on the failed bin, and if we then select another replica number s, then the request will go with probability k/N to bin s and with probability $1/N$ to one of bin k, bin $k+1$, …, bin $N-1$. Thus, if we select the second replica number at random, the request will be served by a certain bin with equal probability $1/(N-1)$. With other words, our system divides load equally in the presence of an inaccessible server.

If we want to lower the total number of containers, we remove the last one, undoing the last expansion. This gives optimal reallocation traffic, where each remaining bin receives traffic worth $1/(N-1)$ of the total size of the system.

The solution is more involved if we are forced to remove an arbitrary container. Assume that container bin i has to be removed. We rename the last added container, bin $N-1$ to bin i. A portion of $1/(N-1)$ of the contents of the former bin $N-1$ came from bin i and stay there. The former bin $N-1$ sends $(N-2)/N-1)$ of its contents to the other containers in equal shares. As the new bin i, it needs to recoup the other $(N-2)/(N-1)$ of the contents of the old bin i, namely those that were not sent to the then new bin $N-1$ in the last extension. Of the total contents, we need to ship $\frac{N-2}{N-1} \cdot \frac{1}{N}$ from the old bin $N-1$ to other containers and we need to ship $\frac{1}{N}$ from the other containers to the old bin $N-1$, which has become the new bin i. This means we move $\frac{2N-3}{N(N-1)}$ of the contents. While not optimal by a factor of $2 - \frac{3}{N}$, at least the work is equally distributed over the remaining containers due to our previous observation.

In order to use our scheme, our resource identifiers need to yield all possible values for the digits in the factorial representation we use. Since the digits correspond to the containers, this gives a relationship between the maximum number of bins and the number of binary digits in the RID (Table I). If RIDs are hash values of unique identifiers, then the maximum number of containers is relatively small, but sufficient for the purposes where we propose our scheme instead of the more involved versions in the literature.

TABLE I
RELATIONSHIP BETWEEN MAXIMUM NUMBER OF BINS AND LENGTH (NUMBER OF BINARY DIGITS) OF THE RECORD IDENTIFIER (RID).

Maximum Number of Bins	Binary Digits in RID	
12	32	
20	64	
34	128	(MD5)
40	160	(SHA-1)
57	256	

One simple possibility to overcome the size limitation is to use the RID in order to seed a good, but cheap random number generator, such as a Mersenne twister or a simple congruential random number generator with a Bays-Durham shuffle [1], [11] in order to generate more digits pseudo-randomly.

V. EXTENSION TO ARBITRARY SIZES

We use the factorial representation of RIDs as a way to encode the decision when to move a recourse to a new container. We can extend this method (at the cost of additional calculations) to containers of arbitrary sizes. Assume that the sizes are s_0, s_1, s_2, ... We mimic the calculation of the factorial digit by using hash functions. We observe that we need to move objects from the existing containers to a new container, bin N, with probability

$$p_N = \frac{s_N}{\sum_{\nu=0}^{N} s_\nu}.$$

If this is the case, then our scheme and its properties hold. We assume that we have k replica of each object and that the first k containers (bins 0 to $k-1$) have equal size.

To organize the moves, we associate the RID of an object to a *decision list* L as follows. We use the RID as the seed of a random number generator and assume that the generator yields the random number sequence $(r_k, r_{k+1}, \ldots)$. We partition the unit interval $[0,1]$ into two parts $[0,p_k)$ and $(p_k,1]$ and the left part into k equally sized subparts, giving us a partition (decomposition into mutually non-intersecting subsets)

$$[0,\frac{p_k}{k}) \sqcup [\frac{p_k}{k},\frac{2p_k}{k}) \sqcup \ldots \sqcup [\frac{(k-1)p_k}{k}, p_k) \sqcup [p_k, 1].$$

If a_k falls into the first interval, we put 0 into L, if it falls into the second interval, we put 1, and if it falls into the last interval, we put k. We call this the quasi-digit x_k. We similarly proceed with $k+1$, generating a list of quasi-digits $L = [x_k, x_{k+1}, \ldots]$.

We then apply our scheme using the quasi-digits in L instead of the digits of the factorial representation. If we have N containers, we place the replica r of an object with RID R into the container b if the maximum index of the digit r in the list L generated by R up to and including index $N-k$ is b, or, if the quasi-digit r does not appear, the replica is stored in bin r. The same analysis as before applies.

VI. CONCLUSIONS

We have presented a general solution to the problem of evenly distributing k replicas of N objects among n bins of equal size. Our solution assumes that all objects are identified by a random resource ID and makes all its allocation decisions based on the factorial representation of that resource ID. Its outcome is a replica-to-bin mapping such that (a) each bin receives the same number of objects, (b) two replicas never share the same bin, and (c) the task of reconstituting the contents of a lost bin is equally distributed among the surviving bins. In addition, our technique handles gracefully any change in the number of bins. Since computing the factorial representation of an integer is a fairly simple operation, there is no need to store these representations anywhere.

In addition, we have presented two extensions of our methods. A first extension eliminates any restriction on the maximum number of bins N. The second applies to bins of arbitrary size.

REFERENCES

[1] C. Bays and S. Durham, “Improving a poor random number generator,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 2, no. 1, pp. 59–64, 1976.

[2] A. Brinkmann, S. Effert, C. Scheideler *et al.*, “Dynamic and redundant data placement,” in *Distributed Computing Systems, 2007. ICDCS’07. 27th International Conference on*. IEEE, 2007, pp. 29–29.

[3] A. Brinkmann, K. Salzwedel, and C. Scheideler, “Compact, adaptive placement schemes for non-uniform requirements,” in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2002, pp. 53–62.

[4] Y. Chen, R. Katz, and J. Kubiatowicz, “Dynamic replica placement for scalable content delivery,” *Peer-to-Peer Systems*, pp. 306–318, 2002.

[5] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong, “Extendible hashing — a fast access method for dynamic files,” *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 3, pp. 315–344, 1979.

[6] R. Honicky and E. Miller, “Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution,” in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. IEEE, 2004, p. 96.

[7] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web,” in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997, pp. 654–663.

[8] C.-A. Laisant, “Sur la numération factorielle, application aux permutations,” *Bulletin de la S. M. France*, vol. 16, 1888.

[9] W. Litwin, M. Neimat, and D. Schneider, “RP*: A family of order preserving scalable distributed data structures,” in *Proceedings of the International Conference on Very Large Data Bases*, 1994, pp. 342–342.

[10] ——, “LH*, a scalable, distributed data structure,” *ACM Transactions on Database Systems (TODS)*, vol. 21, no. 4, pp. 480–525, 1996.

[11] M. Matsumoto and T. Nishimura, “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[12] C. Plaxton, R. Rajaraman, and A. Richa, “Accessing nearby copies of replicated objects in a distributed environment,” *Theory of Computing Systems*, vol. 32, no. 3, pp. 241–280, 1999.

[13] C. Schindelhauer and G. Schomaker, “Weighted distributed hash tables,” in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2005, pp. 218–227.

[14] S. Weil, S. Brandt, E. Miller, and C. Maltzahn, “Crush: Controlled, scalable, decentralized placement of replicated data,” in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 122.