A Bit-Window based Algorithm for Balanced and Efficient Object Placement and Lookup in Large-scale Object Based Storage Cluster

Renuga Kanagavelu
*A*STAR Data Storage Institute, Singapore*
Email:renuga_KANAGAVELU@dsi.a-star.edu.sg

Yong Khai Leong
*A*STAR Data Storage Institute, Singapore*
Email:YONG_khai_leong@dsi.a-star.edu.sg

Abstract

Business requirements for data availability, survivability, and performance have driven the need for building the network storage that interconnects various kinds of storage devices to allow remote access by multiple hosts. A new revolutionary storage technology called "Object based Storage Devices (OSD)" is now emerging as a promising technology to meet the high performance needs and to address various local storage issues. At the same time, it poses several challenges including efficient object placement and lookup in the dynamically changing storage resource environment. We develop a novel and efficient method based on Bit-Windows for object placement and lookup services. We demonstrate the effectiveness of our method through theoretical analysis and simulations results.

1. Introduction

Large-scale Object-based storage systems provide a cost effective and scalable platform for data-intensive applications such as multimedia databases, e-commerce and web crawling. Cluster-based object storage systems provide high performance for storage services. It is highly desirable that a cluster is scalable supporting incremental expansion and high availability. Recently, there has been significant interest in using object-based storage as a mechanism for increasing the scalability of storage systems.

Some key issues that need to be addressed in the design of object-based storage cluster [1, 2] are efficient object placement and lookup and balanced distribution of objects in the dynamically changing storage resource environment. In a large scale object -based storage cluster, object-based storage nodes may be added or temporarily out of reach in a cluster due to node failures. When more nodes are added, objects may need to be relocated from one node to another node to maintain the overall balance of the storage load. It is desirable that such relocations are carried out with low number of object migrations.

Chord [3] uses a variant of consistent hashing to assign keys to Chord nodes. It has the following weaknesses: The locations of objects are not controllable to balance storage usages on different nodes. Apart from this, each node in Chord maintains information about $O(\log N)$ other nodes where N is the total number of nodes, and resolves lookups via $O(\log N)$ messages to other nodes. Bloom filters [4] have been used in OceanStore [5] and tapestry. The weakness of Bloom filters is that the memory and bandwidth required maintaining these filters could still turn out to be too high for a large-scale object based storage cluster. The differentiated object placement and location protocol proposed in [6] combines consistent hashing and bloom filters to take advantage of both of these methods. It uses consistent hashing to locate small objects and Bloom filters to locate large objects.

We develop a new algorithm based on the concept of Bit-Window for providing object placement and look up services. Our method has several attractive features: (1) Objects are evenly distributed over the storage nodes; (2) Placement and lookup operations do not hop through multiple nodes thus reducing the message overhead and access delay; (3) It supports node addition and deletion with low number object migrations; (4) It is scalable; and (5) There is no need to maintain information about neighbor nodes thus reducing message overhead. Our approach focuses on good storage load balance and guarantees storage utilization while ensuring scalability, availability and flexibility.

2. Bit-Window based Algorithm

In this section, we describe the working of the algorithm and explain how it supports various operations: placement, lookup, node addition, and node deletion. We also provide the theoretical analysis of the algorithm.

2.1. Bit –Window

The Bit-Window algorithm maps objects to storage nodes. Let N be the number of nodes with index from 0 to $N-1$. The node indices and the corresponding node IP addresses are maintained in a table at Meta data sever (MDS). The objects are translated into m-bit identifier using a base hash function SHA-1[7]. We call the identifier as object key.

The Bit-Window method maps object key to a node index. The hashed m-bit object identifier is a fixed size bit string .The m-bit identifier is divided into a number of bit windows of size $k = \lceil \log_2 (N) \rceil$. The bit windows are labeled starting from 0 from the right end, denoted as BW_0, $BW_1 \ldots BW_{v-1}$, where v is the maximum number of windows used. In our algorithm v is a control parameter. We note that the size of the bit window depends on the number of the nodes. However, as explained later, due to addition or deletion of nodes, bit window size may vary and our algorithm deals such situations with low number of object migrations.

2.2. Object Placement

Let the object key be represented as

$$b_{m-1}b_{m-2} \ldots\ldots\ldots\ldots b_k \ldots b_2 b_1 b_0$$

The bit-windows of an object key are examined one by one starting from BW_0. If the value of BW_0 is valid then the object key is placed in the node whose index is equal to that value. We note that out of 2^k possible values of a bit window only N are valid and the remaining values are invalid. If the value of BW_0 is not valid, then we continue to examine BW_1, $BW_2 \ldots\ldots\ldots$ BW_{v-1} until a valid node is found. If no valid node index is found among the v windows, the object key is placed at the node whose index is obtained by inverting the most significant bit (MSB) of BW_0. Thus each node stores two sets of objects. Node i is the owner of one set of objects who's BW_0 value is i. The other sets of objects kept in node i are owned by the invalid nodes and node i act as the proxy for them.

Figure 1(a) illustrates the Bit-Window algorithm with N=5. The object key is 011…011111110. The size of the Bit-Window is given by k=3. The value of bit window BW_0 is 6 which is not a valid node index. So the search continues with BW_1 as shown in Fig. 1 (b). The value of the BW_1 is 7 which is also not a valid index. So, the search continues with BW_2 as shown in Fig. 1 (c). The value of BW_2 is 3 which is a valid index and hence the object is assigned to node 3. In this case, the object key is stored in the proxy set at node 3.

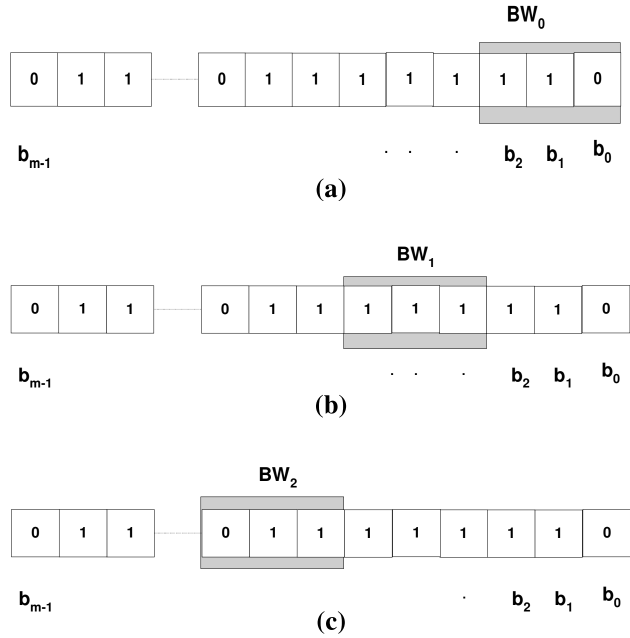

Figure 1. Illustration of Bit–Window algorithm. (a) BW_0 is searched. (b) BW_1 is searched. (c) BW_2 is searched.

2.2.1. Theoretical Analysis

We now show that the objects are evenly distributed by the bit-window algorithm. Let X be the total number of objects. Let N be the number of (valid) nodes and k is the size of a bit-window. Let $P = 2^k$. Let $R = P - N$ be the number of invalid nodes. We note that $R < N$. We assume that object identifiers are mapped to object keys uniformly randomly. For instance, when SHA-1 hashing is used object keys are randomly mapped to a space of 160-bit binary numbers.

When a bit window is examined, its value is a valid node index with probability $\frac{N}{P}$ and is an invalid node index with probability $1 - \frac{N}{P} = \frac{R}{P}$. We say that a bit-window is mapped to an invalid node with probability $\frac{R}{P}$. The probability that none of v windows maps to a valid node index is given by $\left(\frac{R}{P}\right)^v$. Thus, the number of objects which have not yet been mapped to a valid node index is given by $X\left(\frac{R}{P}\right)^v$ and the number of objects mapped to

valid nodes is given by $X - \left[X\left(\frac{R}{P}\right)^v \right]$. Since during each iteration of the algorithm a bit-window is mapped to each of a valid node with the same probability, same number of object keys are mapped to each of the valid nodes and is given by $\frac{X - X\left(\frac{R}{P}\right)^v}{N}$. Similarly, $X\left(\frac{R}{P}\right)^v$ objects are uniformly distributed to R invalid nodes with $\frac{X\left(\frac{R}{P}\right)^v}{R}$ objects mapped to each of the invalid nodes. At this stage, the bit-window algorithm maps each invalid node-index i to valid node index $i - 2^{k-1}$. As a result, among N nodes, each of R nodes will have $\frac{X - X\left(\frac{R}{P}\right)^v}{N} + \frac{X\left(\frac{R}{P}\right)^v}{R}$ objects and the remaining $N - R$ nodes will have $\frac{X - X\left(\frac{R}{P}\right)^v}{N}$ objects. We note that $R < N$ and $N + R = P$ and hence $\frac{R}{P} < 0.5$. As the value of v tends to be large, the term $\left(\frac{R}{P}\right)^v$ becomes negligibly small, and each node will have approximately $\frac{X}{N}$ objects implying a balanced object distribution. Theorem-1 follows from the above analysis.

Theorem-1: The bit-window algorithm evenly distributes the objects to nodes with a node having at most $\frac{X\left(\frac{R}{P}\right)^v}{R}$ additional number of objects than any other node and this number becomes negligibly small when the value of v tends to be large.

Example: Let X = 100 million and N = 1100. This implies that k = 11, P = 2048, and R = 948. Let v=10. Therefore, the number of additional objects a node will have is approximately equal to 50. Among 1100 nodes, each of 948 nodes will have approximately 90916 objects and each of the remaining 152 nodes will have approximately 90866 objects where the difference is only 0.00005%.

2.3. Object Lookup

The object lookup operation of the bit-window algorithm is similar to the object placement. Given the object key, we determine node index using the bit-window algorithm. The corresponding node's IP address is obtained from the table which is maintained at MDS.

2.4. Node Addition

When the new node joins the cluster, it is assigned the next highest index N. We first discuss the case where node addition does not change the bit-window size, i.e., $N < 2^k$. Now, those objects which are meant for the new node will need to be migrated from other nodes. Each of N nodes examines the set of proxy objects and migrates those objects which are meant for the new node. We identify two cases. Each node i examines the objects in its proxy set performs the followings:

1. Those nodes whose BW_0 is N will be migrated.
2. Those objects in which N appears in a bit-window to the right of the bit window with the value i.

Assume that, v is large and each of the existing nodes has $\frac{X}{N}$ objects. When object keys are random and large numbers of objects are dealt with, the migration will lead to even distribution of objects among the $N+1$ nodes with each node having approximately $\frac{X}{N+1}$ objects. We note that objects are migrated from the existing valid nodes to the new node and no object is migrated between the existing valid nodes. Therefore, the number of objects migrated is approximately $\frac{X}{N+1}$ which is the minimum required for even distribution.

We now discuss a special case when an addition of a node needs the bit window to expand. This occurs when $N=2^k$ and each node has only the owner set. In this case, the node index is represented as a $k+1$ bit binary number where k is the original size of the bit window. A node retains the object keys in its owner set such that bit k in BW_0 is 0 and it will consider other objects in its owner set for possible migration.

In the worst case, each node (among N nodes) has approximately 50% of the objects whose bit k in BW_0 is 1. Therefore, in the worst case each node will migrate $\frac{X}{2N}$ objects. Therefore, the total number of objects to be migrated is $\frac{X}{2}$, in the worst case. We note that a change in bit window size occurs infrequently, because, 2^k new nodes should be added in order to increase the bit-window size from k to $k+1$.

2.5. Node Deletion

We identify two cases in node deletion where the size of bit window size does not change as a result of deletion. We note that the case when the size of the bit window needs to be reduced can be dealt with as done in the case of node addition.

Case 1: When a node with the highest index is deleted, i.e. the node with index *N-1* is deleted, we do the following.

The object keys originally stored in node *N-1* will be migrated to other nodes. Each of the object keys is examined using the bit-window algorithm to determine the valid node to which the object key has to be migrated. As a result, approximately $\frac{X}{N}$ objects are migrated from node *N-1* to other nodes. Since object keys are randomly distributed, the number of objects is large, for a large value of *v*, the migrated objects will be uniformly distributed when the bit-window algorithm is used.

Case 2:

When the node other than the highest index node is deleted, we do the following.

Let node *d* be the node to be deleted. The highest index node *N-1* is relabeled as node *d*. All the object keys which were originally in node d should be migrated to this newly labeled node. The object keys which are kept in original node *N-1* are migrated to other nodes the bit-window algorithm. As a result, approximately $\frac{2X}{N}$ objects are migrated. Since object keys are randomly distributed, the number of objects is large, for a large value of v, the migrated objects will be uniformly distributed when the bit-window algorithm is used.

3. Performance Study

We generate the object keys using SHA-1 algorithm. The object keys are 160 bits long. In our experiments *v* = 10. We carry out experiments on two kinds of clusters. The first kind is a small cluster with 9 to 16 nodes and the other one is a large cluster with 100 to 2500 nodes. We consider 5 million objects for placement. We repeat the experiment 10 times to get accurate values with small confidence interval.

We use three important performance metrics in our study. They are (i) number of objects stored at a node, (ii) load imbalance index, and (iii) fairness index. The first metric is self explanatory. The metric 'Load imbalance index' is used to find how badly the most-heavily loaded node is loaded relative to the most-lightly loaded node. Let L_{max} be the number of objects stored at the most heavily loaded node and L_{min} be the number of objects stored at the most lightly loaded node. Then the load imbalance index is calculated as $\frac{L_{max} - L_{min}}{L_{max}}$.

The metric 'fairness index' is used to determine how fairly (or evenly) objects are distributed over all the storage nodes in a cluster. Let *m* and *s* be the mean and standard deviation. Then the fairness index [8] is calculated as $\frac{1}{1+\left(\frac{s}{m}\right)^2}$.

3.1 Performance Results for small storage clusters

Figure 2 shows how evenly our bit-window algorithm distributes the objects across the nodes in a storage cluster with 11 nodes. It plots the number of objects stored at each of the 11 nodes. From the figure, it can be observed that the objects are evenly distributed and the difference between the heavily-loaded node and the lightly-loaded node is only 3249 which is a very small number when compared with the number of objects stored.

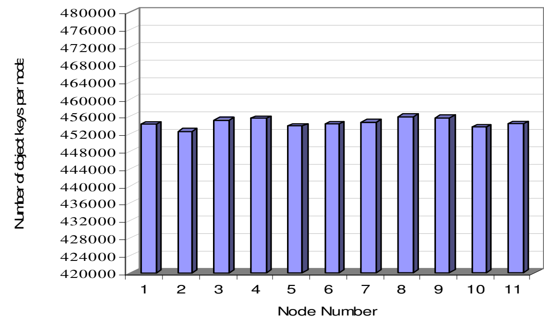

Figure 2. Bit–Window Algorithm: Object Placement in a 11-node cluster

Figure 3 plots the load imbalance index versus the number of nodes when the number of nodes is varied from 9 to 16. From the results, it is observed that the imbalance index is very small. Even the highest imbalance factor itself is below 0.008 which is very small. The highest imbalance factor occurs when N=9 where there are relatively high number of invalid node indices.

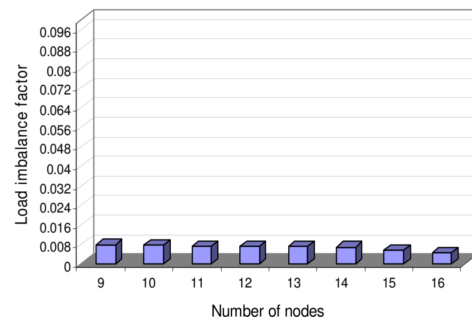

Figure 3. Bit–Window Algorithm: Load imbalance Index for varying number of nodes (small clusters)

Figure 4 plots the fairness index for clusters with number of nodes varying from 9 to 16. We observe that the fairness index is very close to 1. This shows that the bit-window algorithm is very efficient for evenly distributing the objects.

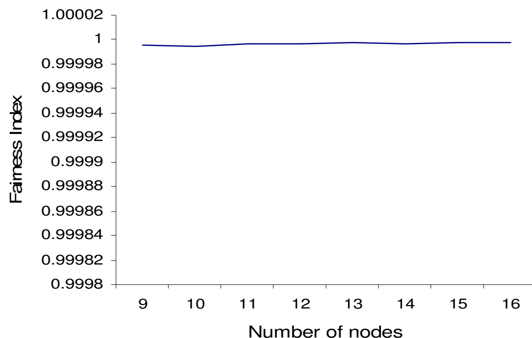

Figure 4. Bit–Window Algorithm: Fairness Index for varying number of nodes (small clusters).

3.2 Performance Results for large storage clusters

Figure 5 shows how evenly our bit-window algorithm distributes the objects across the nodes in a storage cluster with 2500 nodes. It plots the number of objects stored at various nodes. From the figure, it can be observed that the objects are evenly distributed and the difference between the heavily-loaded node and the lightly-loaded node is only 140 which is small when compared with the number of objects stored.

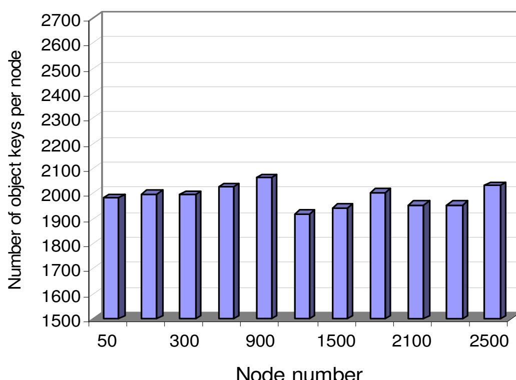

Figure 5. Bit–Window Algorithm: Object Placement for a 2500-node cluster.

Our results show that imbalance factor is very small and fairness index is very close to 1. Due to space limitations, these results are not presented.

4. Conclusions

In this paper we developed a novel and efficient method based on Bit-Windows for object placement and lookup services in object-based storage clusters. Our method ensures even distribution of objects. It does not need to hop through multiple nodes for object placement and lookup, thus reducing the message overhead. It supports node additions and deletions with low number of object migrations. We studied the performance of the method through theoretical analysis and simulation results.

References

[1] Mike Mesnier, Carnegie Mellon and Intel, Gregory R. Ganger, Carnegie Mellon , Erik Riedel, Seagate Research , Object-based Storage, *IEEE Communications Magazine, v.41 n.8 pp 84-90, August 2003.*

[2] Jie Yan ,Yao Long Zhu, Hui Xiong, Renuga Kanagavelu , Feng Zhou ,Lih Weon So. A design of Metadata server cluster in large distributed object-based storage. Twelfth NASA Goddard /Twenty-First IEEE Conference on Mass Storage Systems and Technologies (NASA / IEEE MSST2004) ,April 2004.

[3] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7):422–426, 1970.

[5] 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 International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, November 2000.

[6] Hong Tang and Tao Yang. Differentiated Object Placement and Location for Self-organizing storage Clusters. Technical Report 2002-32, University of California, Santa Barbara., November 2002.

[7] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995.

[8] R. Jain, D. Chiu, and W. Hawe, "A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems," DEC Research Report TR-301, Sep 1984.