Scalable, Structured Data Placement over P2P Storage Utilities

Zheng Zhang[1], Mallik Mahalingam[2], Zhichen Xu[3] and Wenting Tang[3]

Abstract

P2P overlays offer a convenient way to host an infrastructure that can scale to the size of the Internet and yet manageable. Current proposals, however, do not offer support for structuring data, other than assuming a distributed hash table .In reality, both applications and users typically organize data in a structured form. One such popular structure is tree as employed in a file system, and a database. A naïve approach such as hashing the pathname not only ignores locality in important operations such as file/directory lookup, but also results in uncontrollable, massive object relocations when rename on a path component occur.

In this paper, we investigate policies and strategies that place a tree onto the flat storage space of P2P systems. We found that, in general, there exists a tradeoff between lookup performance and balanced storage utilization, and attempts to balance these two requirements calls for intelligent placement decision.

1 Introduction

With the rapid growth of the Internet and ever-rising application's demand, building a scalable and manageable infrastructure is becoming very important. The characteristics of peer-to-peer (P2P) overlay networks in general offer a convenient way to host an infrastructure that can scale to the size of the Internet and yet manageable. The current proposals (e.g., OceanStore [1] PAST [2], Chord [3], CAN [4]), unfortunately, do not directly offer support for structuring data, other than assuming a distributed hash table.

In reality, applications and users typically organize data in a structured form. One dominant form of structured data is *tree*. Instances of tree can be seen in hierarchical namespace used by file systems, and Btrees in databases and Tries for indexing XML documents over the Internet. Even storage system such as semantic file system (SFS), which provides associative access to the system's contents, organizes information in hierarchical fashion. In SFS for example, the queries, scope of the queries, and query results are organized via virtual directories whose structure are very dynamic. In addition, it has been shown that distributed data structures are important for building scalable distributed applications, e.g. for supporting distributed collaboration in the domains such as CAD, protein folding, and genome research, or even for maintaining metadata for managing the system itself.

A naïve approach to store structured data on the existing overlay would distribute objects randomly *anywhere* over the overlay network. While simple, this approach neglects the performance aspect. Although caching can be used to improve the performance, cache-misses can result due to dynamics in the structure of the data and node membership in the system. Furthermore, for a tree, a straightforward implementation of hashing pathname to place objects randomly will result in massive object relocation when changes to the structure occur.

In this paper, we investigate policies and strategies to intelligently place tree onto the overlay to speedup the data traversal by taking advantage of the locality exhibited in most frequent operations. We try to achieve a balance in meeting the following conflicting goals:

- **Performance**: high frequency operations must be delivered with great efficiency. Further, the time to decide where to place an object should also be reasonable.
- **Resource balance**: Efficient resource (e.g. storage) utilization in large-scale network is important, since bad resource utilization may create "hot spots" and system imbalance that can degrade performance.
- **Robustness**: tree shape and construction order should have a negligible impact to both performance and resource balance.

In this paper, we show that simple heuristics are effective in meeting these goals. We propose three algorithms: *radius-delta, hill-climbing* and *zoom-in*. While these algorithms are primarily designed for placing tree, we believe they are generic enough to handle layout for other structured data objects.

As a case study, we use traditional file system to evaluate our approaches. In file system studies, it has been shown that lookup requests comprise a surprisingly high portion of total metadata operations [7]. Moving towards large-scale deployment, we envision this pattern to continue. As a matter of fact, NSFv4 [8] explicitly tries to address this issue by providing multi-component lookup. Optimizing lookups maybe even more important in distributed file systems using P2P overlay, since operations such as create is proceeded by a lookup to the parent directory, after that the actual creation can bypass normal routing infrastructure and operate on the parent object directly. In our study, we choose CAN as the overlay platform.

Our simulations show that our approaches can easily cut down more than half of the lookup costs from the naïve random placement. Some of our algorithms yield even better resource utilization than the pure random policy and are also robust to tree shapes and construction order.

In the remainder of the paper, Section 2 gives the background of the study. We then discuss the three approaches for lookup optimizations in Section 3. Detailed evaluation is offered in Section 4. Section 5 discusses a few orthogonal optimizations. Related works are covered in Section 6 and we conclude in Section 7.

[1] Microsoft Research Asia, zzhang@microsoft.com
[2] VMWare, mallik@wmware.com
[3] Hewlett Packard Laboratories, {zhichen, wenting}@hpl.hp.com

2 Background

2.1 Overview of P2P systems and CAN

P2P systems we are interested in are those that will guarantee the retrieval of an existing object, as opposed to systems such as Freenet [9]. Important flavors of such systems include CAN [4], Pastry [2], Tapestry [1] and Chord [3]. These systems implement a distributed hash: lookup an object is equivalent to searching with the key associated with the object. Similarly, the concept of hashing bucket is mapped to a node in the system. Consequently, object query becomes routing in the overlay network composed by the participating nodes (buckets in the hash). The performance of a query is the product of number of routing hops taken in the overlay network (we call them *logical* hops) and the latency per logical hop. Each logical hop may compose multiple IP-level *physical* hops. Let N be total number of nodes in the system, then for any random pair of nodes, the number of logical hops is a function of N, denoted as $F(N)$.

The system we choose to evaluate is CAN. CAN organizes the logical space as a d-dimensional Cartesian space (a d-torus). The Cartesian space is partitioned into zones, with one or more nodes serve as owner(s) of the zone. Routing from a source node to a destination node boils down to routing from one zone to another in the Cartesian space. The routing cost, $F(N)$, between a random pair is $d/4 \times N^{1/d}$ logical hops. We assume that nodes populate the CAN logical space randomly; this is the default policy in CAN.

2.2 Namespace organization on top of P2P

There are various alternatives to build the namespace on P2P. The most straightforward option would be to hash the entire pathname of an object into a random key, and use it for placement and retrieval directly. For instance, in the case of CAN, */a/b* would hash to point $p_{/a/b}$, and */a/b/c* hashes to point $p_{/a/b/c}$, and so on so forth. Locating these objects then amounts to routing to the corresponding points in the logical space. This does not require any change to the underlying infrastructure. However, this works best for immutable namespace in practice. Consider that the path component *b* is changed to *b'*. This renaming operation not only renders that the directory */a/b'* be hashed to a different point and thus has to be relocated physically, but *all* pathnames following the directory *b* (now *b'*) are affected as well and therefore their corresponding objects have to be moved. Figure 1 helps to explain this phenomenon.

The problem here is that the placement decision creates binding with the structure of the namespace. Other hashing alternatives, such as hash on the content of directory objects, have similar (albeit reduced) problem.

Therefore, to have a namespace hierarchy that can cope with structural change, each directory must contain name of the children object as well as their location info (point in the Cartesian space in the case of CAN). This is very much like the directory structure in conventional file system, in which inode number is the routing key for the layers beneath. When location information is embedded, the placement of the objects becomes controllable, allowing us to explore locality in various namespace operations. (This is demonstrated in Figure 2, where the binding between a parent directory and all its sub-directories can be explored in *recursive* namespace lookup to reduce overall lookup cost.)

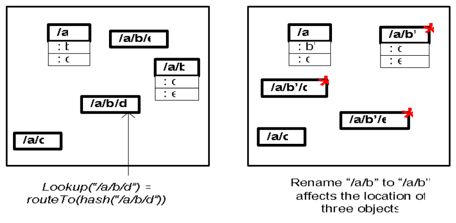

/a
: b
: c
/a/b/e
/a/b
: d
: e
/a/b/d
/a/c
Lookup("/a/b/d") = routeTo(hash("/a/b/d"))
/a
: b'
: c
/a/b'
: d
: e
/a/b'/d
/a/b'/e
/a/c
Rename "/a/b" to "/a/b'" affects the location of three objects

Figure 1: Using hashing on pathname alone can cause massive, uncontrollable relocation when rename occurs (objects with * attached are affected)

2.3 Lookup operation in the namespace

The primary function of lookup is to locate the object, with one of the implicit requirements of validating the path along the way.

Lookup can be implemented in two different ways. The first, *iterative*, lets the client resolve one component at a time. This method is commonly used in local file system [10] in order to resolve over mount points. The other, *recursive*, hands over the complete path to the first directory, which resolves the first component and then passes the remaining path to the next directory, and so on so forth. The final result is then sent back directly to the client. These two approaches are shown in Figure 2. When a distributed system is built on P2P, different directories are very likely to reside on different nodes. Therefore, irrespective of the implementation, the unique directories (and thus unique nodes in the system) to be visited are the same. Obviously, the recursive approach reduces the total number of network hops and is more adequate in large-scale system where latency is high. To lookup an object N levels down, iterative requires $2N$ network trips, comparing with $N+1$ of recursive. For this reason, we choose the recursive method in this report.

2.4 Cost of lookup

We can now discuss how lookup cost is computed.

For a given object, let P be the complete path (e.g., "/a/b/c"), and $D=|P|$ be the length of the path. The total lookup cost can be expressed as $D \times \bar{h}$, where $\bar{h}$ is the number of average logical routing hops resolving *one* component. Assuming random node distribution in the logical CAN space, for any two nodes with the same logical distance, the physical routing latency between the two is roughly the same. As a result, we can compare various algorithms by their total logical routing hops. The two simple cases of lookup cost, each represents one extreme of the cost spectrum, are as follows:

- If we can compute the location of the object *a priori* (e.g., hashing pathname), the so-called *zero* lookup bypasses all directories on the path and directly seek to the object. Since the query may be submitted from anywhere in the system, the total cost is equivalent to the routing cost between a random pair of nodes in the system. The zero lookup cost is therefore *F(N)*.

- If, on the other hand, we allow any directories on *P* to reside *randomly* anywhere in the system, then resolving any component incurs the cost of routing between two random nodes. Therefore, the total cost is $D \times F(N)$. This solution, which we call *baseline*, has the best storage utilization.

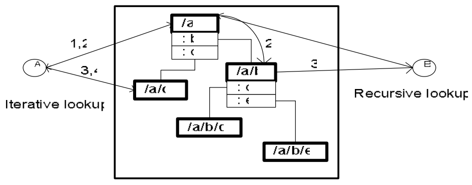


Figure 2: Two lookup procedures shown side by side. Node A performs iterative lookup, while node B performs recursive lookup. Each directory may reside on different nodes in the system. Links between objects are location information embedded in the directory entries. As shown, iterative incurs more network hops than recursive.

F(N) and $D \times F(N)$ represent the low and high bounds of lookup cost. Throughout the rest of the paper, we use L_b and L_0 to denote the cost of baseline and zero lookup, respectively. A good solution should achieve lookup cost as close to *F (N)* as possible, while keeping the storage utilization close to that of the baseline. In addition, the algorithm must be simple and efficient. This is especially important for large-scale peer-to-peer systems.

2.5 Discussion

It should be noted that in traditional distributed file system, path resolution results are often cached and thus lookup cost is paid only once. This has been particularly effective for small-scale system, where namespace structure does not change rapidly and/or the update cost is small. For a large scale P2P system, however, cached directory entries may get invalidated not only because the structure change, but also when their location (zone) change (i.e. the node hosting the original directory object). Furthermore, cache misses can be more expensive. What this means is while we still expect caching namespace resolution to be helpful, a system must not rely on caching alone but rather to start with a sound placement strategy to begin with.

3 The algorithms: *Radius-delta*, *Hill-climbing* and *Zoom-in*

As explained earlier, the total lookup cost can be expressed as $D \times \bar{h}$, where *D* is the path length, and $\bar{h}$ is the number of average logical routing hops resolving one component. The locality of the lookup procedure is inherent in resolving consecutive components. Therefore, to bring down $\bar{h}$, we can place a child object at a node that is close-by in routing with respect to the node hosting the parent object, subject to storage utilization constraint and decision complexity. This is the principle behind the *Radius-delta* and *Hill-climbing* algorithms. The third algorithm, *Zoom-in*, takes a somewhat different approach. The goal here is to quickly "zoom" into a small subset of nodes that host deep down sub-trees, making lookup cost irrelevant to the path length *D*.

3.1 Radius-delta

The idea here is to simply choose a small constant, *r*, which defines a small distance within which a child object will be randomly placed, relative to the position of the parent. In the case of CAN, *r* is a small real number in the range of [0,1], for example 1/16.There is no additional overhead in creation time: it is just a matter of picking a random point in the target space.

Under this algorithm, the average distance between a parent and one of its children is *r/2*. The routing cost resolving one "component" is $r \times F(N)$. As a result, the total lookup cost is $D \times r \times F(N)$.

With a balanced tree, the storage utilization with infinite number of nodes resembles a normal distribution centered at root. The height of the center depends on *r*: bigger *r* has a flat center and spreads out the distribution. Also, the deeper the tree relative to *r*, the more spread the distribution will be. This seems to imply that radius-delta by default have very poor storage balance. However, the system is not of infinite size. The effect of limited system size is to divide the "infinite" spread into chunks, and the total storage utilization can be found by "folding" the chunks on top of each other. Figure 3 illustrates these concepts. For the "folding" effect to take hold, $D \times r$ must be larger than 1 to allow the allocation "crawl" out of the boundary. Thus, a larger $D \times r$ has the net effect of smoothing up storage utilization, while at the same time increases lookup cost. As a result, there exists a tradeoff between lookup time and storage utilization.

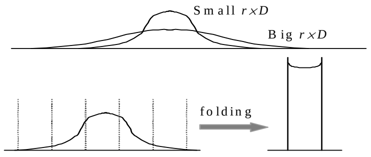


Figure 3: Storage utilization with radius-delta and the "folding" effect

3.2 Hill-climbing

In a P2P network, nodes typically exchange periodic heartbeats with its neighbors for maintenance purposes. Storage utilization of surrounding nodes can be obtained for "free". Guided with this knowledge, a node inspects its own storage utilization comparing with those of its immediate neighbors. If the minimum storage utilization of its neighbors is within a *threshold*, comparing with that of its own, it hosts the object immediately. Otherwise, it hands the job over to the one with the minimal utilization (breaking ties randomly) and the process starts there again. The default algorithm has a threshold of zero. A larger threshold value encourages parent-child collocation, at the cost of less even storage distribution.

Lookup cost depends largely on parent-child distance. Consider the initial state of the system where there is no object, and assume the namespace is built recursively breadth-first. A "pile" will first emerge, with the later-comers laid on the surface. Since the surface increases gradually, the pace at which the "pile" expands also slows down. When the next level of hierarchies starts to build, new "shells" are put up and the "pile" crawls towards outbound. In reality, the actual tree creation order is arbitrary, and the hill-climbing algorithm always tries to place the object close-by, given the constraint of storage utilization. A deep and thin tree usually has short parent-child distance, whereas a fat and shallow tree is the opposite. As a result, lookup cost to leaf objects is insensitive to the tree shape.

The hill-climbing algorithm cannot always settle the placement of the object in one shot. If the parent directory is already in a "dip", the placement is immediate. If, on the other hand, the parent is at the top of a local "hill," the algorithm will roll "downhill" until a "dip" is found. In that case, creation takes longer time. A larger threshold tends to reduce the creation cost because it tolerates utilization deviations more. Another way to limit the creation cost is to introduce the time-to-live (TTL) value to restrict the number of hops the placement takes.

Multiple hills can emerge and, consequently, the total storage utilization becomes uneven. This is so because the optimization is always local, which is a fundamental property of the hill-climbing algorithm. When number of objects is much larger than the system size, the multiple-hill effect will be reduced. This is because the "pile" will eventually "overflow" the boundary and crawl backwards. Figure 4 demonstrates these concepts.

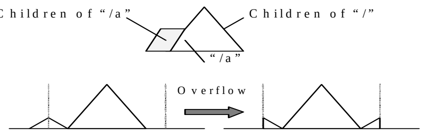


Figure 4: Storage utilization of hill-climbing algorithm

3.3 Zoom-in

While the previous two algorithms aim at bringing down $\bar{h}$, the goal of zoom-in is different. Zoom-in instead tries to minimize the impact of *D* and, in the extreme, renders it completely irrelevant.

With zoom-in, we always define the region where a parent object lives, and sub-divide that region into *k* (*k* is called the *zoom-in degree*) sub-regions in which we place the child objects in a random or round-robin fashion. This way, descending down the tree, the routing range will be recursively reduced, until the point where one single node now contains all the remaining sub-tree of a directory. Assuming sub-regions are perfect cubes and with fully balanced tree, we can prove that this algorithm can approach the performance of L_0:

For a *d*-dimensional CAN, assuming a zoom-in degree *k*, and a region with average routing hops *x*, after zoom-in, the average routing hops of each of the *k* sub-regions will be $x*(1/k^{1/d})$. If we repeatedly sub-divide the sub-regions with the same zoom-in degree, after subdividing the region *y* times, the average routing hops of the sub-regions will be $x*(1/k^{y/d})$. An upper-bound of the average lookup time is therefore, F(N) * $\sum_{y=0}^{\infty}\left(\frac{1}{k}\right)^{\frac{y}{d}}$, which is approximately $[k^{1/d}/(k^{1/d}-1)]\times F(N)$. With a large *k*, zoom-in approaches the performance of L_0.

There is no additional overhead involved in object creation. As long as the namespace tree is balanced, the storage utilization will also be perfect. But there is one catch: if the fan-out is smaller than *k*, then it's guaranteed that not all sub-regions will be populated. This is especially a problem if small fan-out occurs at higher level. Furthermore, zoom-in only works the most effectively if *D* is large (a shallow tree can be easily handled by radius-delta). Therefore, this algorithm is good for deep, balanced tree with fan-out a multiple of the zoom-in degree (especially at higher level). In other words, zoom-in is somewhat sensitive to tree shapes. We discuss remedies to that problem in later sections.

When *a priori* knowledge of the tree is available (for example a digital library), it is possible to do intelligent division of the regions (i.e, vary *k* and sub-regions sizes accordingly). When the shape of the tree changes causing a region for a sub-tree to become over crowded, we can always allocate a new and less crowded region for the new objects of the sub-tree that otherwise would fall into the over crowded area.

3.4 Summary

(1) By choosing small radius, radius-delta can arbitrarily reduce the lookup cost. However, the storage utilization improves only upon the product of the depth of the tree and the radius. As a result, there exists a tradeoff between lookup performance and storage utilization. (2) Hill-climbing tries to minimize lookup cost given storage utilization constraint, and is insensitive to tree shapes. But it may take time to make placement decision, and require large object-to-system size ratio to populate system space.

(3) Zoom-in has the best theoretical performance, with lookup cost approaching that of zero lookup. However, it's more sensitive to tree shapes. Zoom-in with *a priori* knowledge of the tree structure is possible to attain good lookup performance as well as storage utilization.

The design of the above algorithms has taken the three requirements, performance, storage utilization and robustness, into account.

4 Evaluation

We evaluate the three algorithms proposed by means of simulations. The primary goal of the experiment is to gain insight on how well the approaches perform, and secondly to identify potential optimization opportunities.

We choose CAN as the overlay network consisting of 1024 nodes (roughly about 1/100 of total objects for a given a tree), organized in a 2-d Cartesian space. Nodes populate this logic space uniformly. The details of experiment setup, workload and metrics of comparison are reported first, followed by the results and analysis.

4.1 Experiment setup

4.1.1 Namespace used

The namespace trees include both synthetic as well as real, active ones. In the synthetic class, two trees with dramatic different characteristics, *deep* and *fat*, are used. Deep is a binary tree containing objects spanning across 31 levels. The fanout in this tree grows slower, with fanout alternating between 1 and 2 per node as level grows. Fat tree has fan-out that doubles at each subsequent level, with initial fan-out of 4. Fat tree contains 100K objects with depth of 6. We gathered real active trees from two different sources. The first one is a namespace from an active server for software distribution in HP-Labs, and this tree has about 165K objects in total. The second active tree is web namespace that we generated from the proxy logs available from NLANR [11], which contains about 1M objects.

We use *d, f, h and w* as shorthand notations for these four trees from now on. Figure 5 shows the number of children at each level for these trees.

4.1.2 Access pattern

Namespace accesses correspond to two different kinds: those that construct the tree, and those that walk through the tree (lookup).

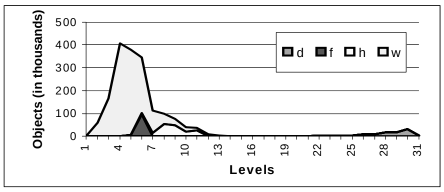


Figure 5: Shapes of the four trees used. (Where d, f, h and w refer Deep, Fat, HPL tree and Web Namespace respectively)

Since the log of tree creation is not available, to understand the effect of different tree creation order, we synthetically construct tree three different ways: depth first (DF), breadth first (BF) and random (R). Since the robustness against tree construction order is by itself an important issue, we dedicate a separate section to report its effect. In other parts of the evaluation, BF is used as the default tree construction order.

Access pattern is synthetic for all but the web namespace. In the case of synthetic access pattern, we randomly choose a set of paths from the processed tree and a set of nodes from the node space, to perform lookup. For the web namespace, the access pattern is directly taken from the proxy logs. We first hash the clients IP address to one of 1K locations and use the hashed value as node identifier to perform the lookup with the corresponding path that client accessed.

4.1.3 Metrics

For a given lookup operation, the cost is computed by counting logical hops routing to leaf objects. This gives us the direct measure of $\bar{h}$ -- number of logical hops resolving one segment in lookup. The number of hops "seeking" to the root is included. We average over 100 random samples; each is a pair of the query node and a leaf object. To give as fair a comparison as possible, we report the lookup cost, *L*, normalized by the zero lookup cost (L_0). The normalized baseline lookup costs (L_b) for the four trees are 27, 5.6, 8.7 and 5.2 for *deep*, *fat*, *hpl* and *web* respectively. Due to the irregularity of node's physical connectivity and position in the Internet, logical hop counts may not be perfectly proportional to the end physical latency observed. However, we believe *L* establishes a sound base for performance comparison in evaluating different algorithms.

The storage utilization is the standard deviation amongst nodes. Similar to lookup performance, the storage utilization, *U*, is normalized by that of the baseline.

Wherever necessary, we also report the creation time *C*, which is the number of hops it takes before an object is finally placed. *C* represents the overhead in performing intelligent placement decision.

4.2 Results

Figure 6 shows the result of the basic radius-delta algorithm. For comparison purposes, we include the data of the baseline algorithm (data points labeled by "b"). We use altogether 5 different radiuses, from 1/16 to 1 (r*d* uses *1/d* as radius). There are clear tradeoffs between lookup cost versus storage utilization: *L* increases, whereas *U* decreases with radius. As described earlier, larger radius and *D* has the effect of "overflow" the boundary and even up the distribution. Consequently, *deep* has the best storage utilization, followed by *hpl*. Because *fat, hpl* and *web* are relatively shallow, lookup performances benefit from small radius for these trees.

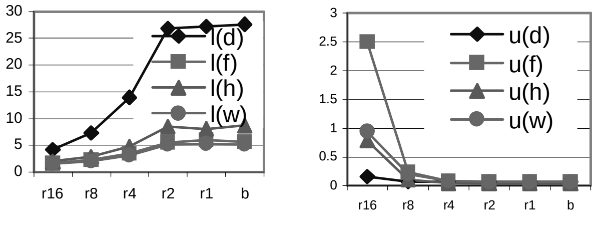
For these workloads, radius of 1/8 is the most balanced between good lookup performance and reasonable storage utilization. On average, its lookup cost is about 4 times of L_0, 67% reduction over L_b.

Figure 7 depicts the result of the hill-climbing algorithm. We also studied variations with threshold and TTL. The threshold is number of objects as a fraction of the average objects per node. For example, if there are 100K objects in a system of 1K nodes, then average objects per node is 100. In this example, a threshold of 5% means a node will host an object unless one of its neighbors has 5 less objects. Because total number of objects is usually unknown at creation time, this threshold setting may not be practical. An absolute storage utilization difference (weighted with bandwidth and other parameters) is more adequate. TTL is simply the maximum number of hops a placement will take before settling down. In Figure 7, *hc* is hill-climbing with infinite TTL; *hc-T* is hill-climbing with TTL equals to 10; *hc+* is hill-climbing with threshold of 5% and infinite TTL; *hc+T* is the same as *hc+* but with TTL=10. As before, baseline results are included.

The first observation to make is that, across the board, all variations achieve good storage utilizations as well as low lookup costs. Interestingly enough, the storage utilization is even better than baseline. It could also be observed that different trees make little effect. This is because for a given number of objects, a fat tree has low *D* and the average parent-to-child distance is greater; a deep tree is exactly the opposite. Setting non-zero threshold produce the same effect. Overall, lookup cost is around 3 times of L_0, achieving 78% reduction over L_b.

This robust performance comes with a slight cost of creation cost, since hill-climbing will look for storage under-utilization nearby. The creation costs for *hc* are 2.2, 4.2, 6.3 and 3.7 hops, for *deep*, *fat*, *hpl* and *web*.

Figure 8 gives the result of the zoom-in algorithm. The zoom-in degrees are is 8, 4, 2 and 1 (z*d* is zoom-in degree equals *d*). Zoom-in degree of 1 *is* the baseline algorithm.

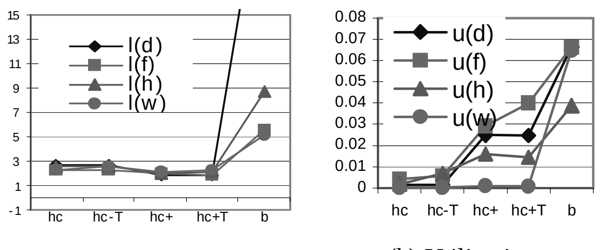


(a) Lookup cost (b) Utilization

Figure 6: Radius-delta. *r*d means using radius delta of 1/r. The baseline results are the points on the "b" column.

Let us focus first on the lookup cost. As expected, zoom-in tends to be robust against different trees. Z4 seems to be good enough for these workloads, achieving lookup cost of 3.4 times of L_0, a 64% reduction over L_b. However, different trees have a significant impact on storage utilization. *Deep* with zoom-in degree of 8, populates a very small fraction of the system and its utilization *is* very poor (*U*=12K). However, as observed in Figure 5, the *web* tree has a very big fan-outs, as a result, storage utilization is excellent. This verifies our assumption that while zoom-in can attain the best theoretical lookup performance, its storage utilization is sensitive to different tree shapes.

4.3 Variations



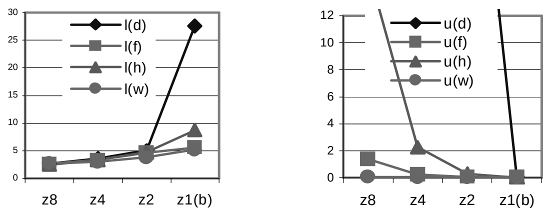
(a) Lookup cost (b) Utilization

Figure 7: Hill-Climbing algorithm.

The hill-climbing algorithm can be a standalone tool to even up distribution. We apply this to radius-delta and zoom-in to see its effects. We use radius-delta and zoom-in as algorithms to make primary placement decision, and use hill-climbing to further fine-tune the placement according the storage utilization in the local context. Doing so may reduce the lookup performance, since hill-climbing will move the placement to nearby underutilized nodes. We set the threshold to be 5%.

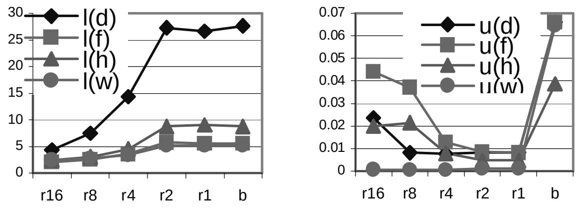
Figure 9 shows the results of radius-delta, when applied with hill-climbing. Comparing with its non-hill-climbing counterparts, the storage utilization is greatly improved. Lookup costs increases are moderate. Since radius-delta has fairly good storage utilization to start with, it does not take hill-climbing too long to smooth the distribution: the average creation cost of r16, for example, is only 2 hops.

Figure 10 shows that hill-climbing is also efficient to improve storage utilization for zoom-in. This comes with the cost of more creation time. Especially in z8 where storage utilization is the most imbalanced, it takes hill-climbing lots of time to even up utilization (creation hops). The impact on lookup is the most pronounced for *deep*: *L* with z8 increases from 2.7 to 8.2.



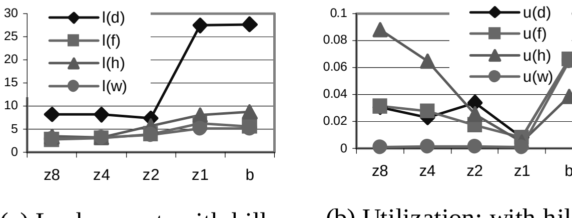
(a) Lookup cost (b) Utilization

Figure 8: Zoom-in algorithm



(a) Lookup cost: with hill-climbing (b) Utilization: with hill-climbing

Figure 9: Radius-delta with hill-climbing



(a) Lookup cost: with hill-climbing (b) Utilization: with hill-climbing

Figure 10: Zoom-in with hill-climbing

To gain more insight of the effectiveness of hill-climbing, Figure 11 plots the storage utilization of the radius-delta algorithm, before and after hill-climbing is applied. In the figure, x-axis corresponds to nodes in the system (node space is compacted into 40 chunks) and y-axis corresponds to the normalized storage utilization. The results are for *fat* tree with the smallest radius we experimented, 1/16. Recall that in radius-delta, storage utilization depends on $D \times r$. Thus, this is the most unfavorable case to start with and the concentration is effectively mitigated after the hill-climbing is applied.

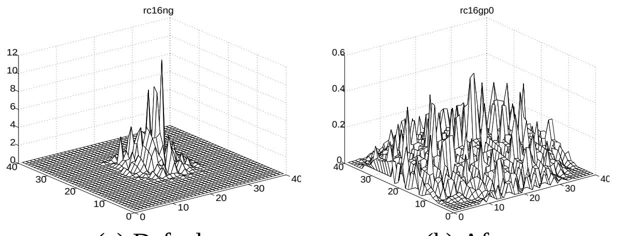
4.4 Robustness against tree creation order

In the above experiments, we have assumed a breath-first tree construction order. In reality, tree creation is far from predictable, not to say controllable. An important aspect of the placement algorithm is therefore how robust they are against different tree creation order.

For tree creation order to make a difference, the placement decision must have a dependency on the current layout status. As such, neither the basic radius-delta nor the zoom-in algorithm is sensitive to creation order: they depend on the overall structure *only*. However, the hill-climbing algorithm itself and the other two algorithms when combined with it are influenced by storage utilization at any given moment. To understand the impact of the tree creation, we use three forms of order, depth-first (DF), breadth-first (BF) and random (R). For random tree creation, we enumerate all the paths of each tree in random order, we then walk each component of the random paths and create an object if the component has not yet been created. We only report results of the following configurations:

- Hill-climbing with threshold of 5 and 10 TTL.
- Radius-delta with *r=1/8*, combined with hill-climbing with threshold 5.

Zoom-in with degree of 4, combined with hill-climbing with threshold 5.



(a) Default (b) After

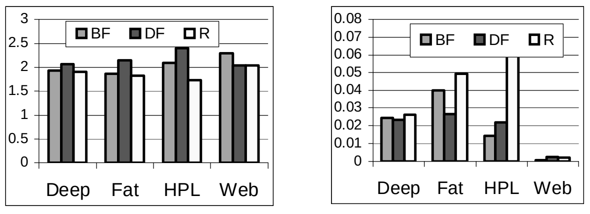
Figure 11: Effect of hill-climbing on storage utilization (radius-delta with r=1/16, on the fat tree).

We plot the lookup cost and storage utilization of DF, BF, and R as shown in Figure 12 through Figure 14. It can be observed that the order of tree creation has very little effect on the lookup cost and storage utilization. For the configuration with the hill-climbing algorithm alone, random tree creation slightly worsens the storage utilization for both the fat tree and the HPL tree. Both the fat and hpl trees are fat and shallow, and of relative small sizes. We believe that for small fat-trees, with a random strategy, there is a larger probability that multiple hills can emerge and consequently cause the total storage utilization uneven, whereas for a larger tree, such as the Web tree, the small hills are evened out because of the huge number of objects. Nevertheless, even the worst case has better storage utilization than the baseline case.

4.5 Summary

In this section, we provide extensive experiment results on the three heuristic algorithms. Not surprisingly, in general, improving lookup performance comes at a cost of less even storage utilization distribution. Algorithms that try to balance the two calls for intelligent placement decision, which incurs additional overhead.

Our results indicate that the simple hill-climbing algorithm is robust and achieves the most balanced results overall, with moderate placement overhead. Hill-climbing is also very effective as a complementary tool to smooth out distribution for other algorithms. Radius-delta delivers fair performance, and its storage utilization can be improved quite easily using hill-climbing. The strength of zoom-in is for deep, balanced trees with fan-outs greater than zoom-in degree, and is demonstrated with the *web* namespace tree, which is shallow and has high fan-outs.



(a) Lookup cost: (b) Utilization

Figure 12: Lookup cost and Storage utilization for hill climbing using BF, DF and R tree creations

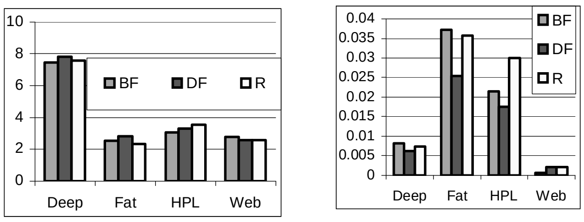
5 Discussion and other mechanisms

We assume that storage distribution is at the granularity of object size. Other alternatives such as CFS [12] choose to treat the whole distributed storage space as a gigantic disk and therefore allocation is done at the block level. The difference in distribution granularity is not the key issue, as resolving next component along the namespace path necessitates the access of component's internal data in any case – be it scattered on a number of blocks or contained in one object. Lookup performance only depends on the cost of seeking from one component to next.

In fact, we believe the algorithms we developed here is directly applicable not only as optimizations to improve lookup performance in block-based distributed file system on top of P2P, but can be extended to data blocks allocation as well. If an object is large, the blocks associated with it may spread across multiple nodes. Distribution of those blocks should take into account the access pattern of the application. If the access pattern isn't predictable, it will be safer to target the locality exhibited in sequential access and/or pre-fetching. Sequential access to a broad range of blocks is akin to "lookup" through a thin tree fan-out equal to 1. As such the blocks should be allocated close by, subject to storage utilization constraint. This is exactly the same assumption under which our algorithms for metadata distribution are developed.

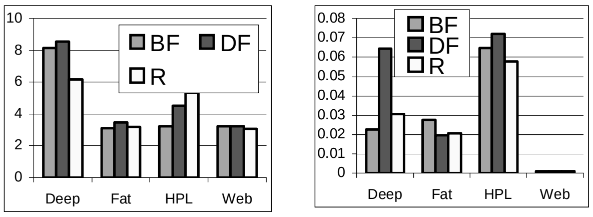
Since lookup performance depends on system's routing capability, a number of obvious optimizations can be applied to shift the focus more towards storage utilization. For instance, we can use high-dimension CAN or other systems with O (log N) routing performance. Each node might have routing cache to keep direct routes to those frequently visited nodes. Alternatively, we can include a "hint" field in directory entry that speculates the node that keeps the child object. Apart from this, other well-known approaches such as caching and replication, directory aggregation, where an object contains multiple levels of underneath sub-directories, will also reduce the lookup cost by slashing *D*.These approaches work well but only for namespace objects that are rarely modified. In a large-scale peer-to-peer system, enforcing consistency for directory objects that need to be accurate all the time is a sizeable challenge. Approaches such as directory aggregation will run into "false-sharing."

Another vulnerable point of the above strategies has to do with the dynamic nature of P2P system. The side effect introduced when nodes come and go may also invalidate cached copies. Consequently, the system can't rely on caching alone, and a sound placement algorithm relying on default routing infrastructure *only* is highly beneficial.



(a) Lookup cost: (b) Utilization

Figure 13: Lookup cost and Storage utilization for Radius-delta (8) using BF, DF and R tree creations



(a) Lookup cost: (b) Utilization

Figure 14:Lookup cost and Storage utilization for Zoom-in (4) BF, DF and R tree creations

6 Related work

Much work has studied storage utilization in large-scale systems. Many of them are variations of hill climbing; this is the case in Farsite [13], CAN[4] and PAST[2]. The hill-climbing algorithm we proposed is guided by the same principle. Our unique contribution is to treat the issue of storage utilization in conjunction with efficient hierarchical namespace in the P2P networks.

CFS [12] proposes an alternative called "virtual server" to divide the resource of a physical server. The net result is that storage utilizations of multiple nodes in the overlay networks are aggregated. If those virtual servers are scattered sufficiently randomly, overall storage utilization in the physical world tends to be balanced. The tradeoff is that more states have to be kept per physical node. The locality of file system operations and its impact on performance was not taken into account.

Several systems are capable of building a distributed file system at object granularity [14, 15]. Lightweight protocols to construct namespace across geographically distributed sites are proposed in [16]]. In [17], qualitative arguments have been made about the gap between the hierarchical namespace and the flat P2P storage abstraction, pointing out that the "virtualization" property of P2P systems is incapable of taking advantage of the various locality exhibited by applications. Our view is consistent with theirs, and we have contributed not only with quantitative analysis but also algorithms to balance the tradeoff between performance and resource utilization.

7 Conclusion and Status

We investigate the issue of placing hierarchical structure over storage space offered by P2P systems.

We found that, there is a tradeoff between lookup performance and uniform storage utilization distribution. To balance the two requirements calls for intelligent placement decision that may incur additional overhead.

We propose three simple algorithms and verify them through simulations. The design goal has been to balance carefully between performance, storage utilization, and robustness. Our approach is application driven and accounts for locality in high-frequency operations.

Our results indicate that the simple hill-climbing algorithm is robust and achieves the most balanced results overall, with moderate placement overhead. Hill-climbing is also very effective as a complementary tool to smooth out distribution for other algorithms. Results show that our approaches can reduce the average lookup cost by 60-70% from the pure random placement.

8 References

1. Kubiatowicz, J., et al. *OceanStore: An Architecture for Global-Scale Persistent Storage*. in *ASPLOS 2000*.

2. Druschel, P. and A. Rowstron. *PAST: a large-scale, persistent peer-to-peer storage utility*. in *HotOS-VIII Workshop*. 2001. Schloss Elmau, Germany.

3. Stoica, I., et al. *Chord: A scalable peer-to-peer lookup service for Internet applications*. in *ACM SIGCOMM*. 2001.

4. Ratnasamy, S., et al. *A Scalable Content-Addressable Network*. in *ACM SIGCOMM*. 2001. San Diego, CA, USA.

5. Gnutella, http://www.gnutella.org.

6. Rowstron, A. and P. Druschel. *Pastry: Scalable, distributed object location and routing for largescale peer-to-peer systems*. in *IFIP/ACM Middleware*. 2001. Heidelberg, Germany.

7. Dahlin, M., et al. *Cooperative Caching: Using Remote Client Memory to Improve File System Performance*. in *Usenix OSDI*. 1994. Monterey, California, USA.: USENIX.

8. NFSv4, http://www.nfsv4.org.

9. Clarke, I., et al. *Freenet: A distributed anonymous information storage and retrieval system*. in *Workshop on Design Issues in Anonymity and Unobservability*. 2000. Berkeley, CA, USA.

10. Kleiman, S.R. *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*. in *Summer 1986 USENIX Conference*. 1986. Atlanta, GA, USA.

11. NLANR, http://www.nlanr.net/.

12. Dabek, F., et al. *Wide-area cooperative storage with CFS*. in *Symposium on Operating Systems Principles (SOSP)*. 2001. Banff, Canada.

13. Bolosky, W.J., et al. *Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs*. in *ACM SIGMETRICS*. 2000. Santa Clara, California, USA.

14. Silaghi, B., B. Bhattacharjee, and P. Keleher. *Routing in the TerraDir Directory Service*. in *In Submission*. 2002: University of Maryland.

15. Karamanolis, C., et al., *An Architecture for Scalable and Manageable File Services*. 2001, Hewlett-Packard Labs.

16. Zhang, Z. and C. Karamanolis. *Designing a Robust Namespace for Distributed File Services*. in *20th Symposium on Reliable Distributed Systems*. 2001. New Orleans, USA

17. Keleher, P. and S. Bhattacharjee. *Are Virtualized Overlay Networks Too Much of a Good Thing?* in *1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*. 2002. Cambridge, MA, USA.