A Novel Load Balancing Mechanism for P2P Networking

Leonidas Lymberopoulos
National Technical University of Athens
9 Iroon Polytechneiou
15780, Athens,Greece
+30 210 772 1448
leonidas@netmode.ntua.gr

Symeon Papavassiliou
National Technical University of Athens
9 Iroon Polytechneiou
15780, Athens,Greece
+30 210 772 2550
papavass@mail.ntua.gr

Vasilis Maglaris
National Technical University of Athens
9 Iroon Polytechneiou
15780, Athens,Greece
+30 210 772 2503
maglaris@netmode.ntua.gr

ABSTRACT

Peer to Peer (P2P) networking is a potential disruptive technology that can be used for the development of scalable, fully decentralized distributed applications. However, to realize its potential, P2P technology should address the needs of a variety of applications, other than file-sharing requiring support for exact-match queries on the file names. Our work complements and contributes to existing P2P overlays that support multiple-attributes and range queries, using the distributed K-Dimensional (K-D) tree structure for organizing shared information among participating peers. This guarantees that the time needed for node join - leave operations and query times are logarithmic with respect to the number of peers. In such systems, an open issue is load balancing of resources among peers, as only load-balanced data structures can guarantee that the complexity for resolving multi-attribute and range queries remains logarithmic (thus scalable) with respect to the number of participating peers. In this paper, we report a novel load balancing algorithm for dynamically keeping the resource load among peers balanced. We prove that the load balancing algorithm is robust and scalable, achieving an $O(\log^2 N)$ complexity, where N is the number of peers. We illustrate how our algorithm can be used to build a scalable Grid Information Service supporting multi-attribute and range queries on available services within the shared Grid infrastructure.

Categories and Subject Descriptors

D.2.1 [**Computer-Communication Networks**]: Network Architecture and Design– Distributed Networks

General Terms

Algorithms, Performance

Keywords

P2P load balancing, Distributed K-D tree, P2P Grid Information Service

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GridNets 2007, October 17-19, 2007, Lyon, France

Copyright 2007 ICST 978-963-9799-07-3

DOI 10.4108/gridnets.2007.2254

1. INTRODUCTION

P2P computing has emerged as a significant technological and social phenomenon over the last years. It provides a scalable and fully distributed system used for locating shared resources within a large number of peers, in the order of tens of thousands. P2P systems have been mainly used for file sharing applications. However, there is an increasing need for P2P support for a variety of application classes that require efficient and scalable resolution of complex queries involving multiple different attributes and possibly range queries, other than finding the exact name of a file [1,2].

As an example Grid Information Services is a class of applications demanding complex P2P queries about shared resources, such as CPU load, storage and memory. Grid nodes and services can dynamically join and leave the system, introducing a dynamic environment, where still one has to be able to query about available resources. In most Grid environments, the Information Service system used for locating resources is centralised, having usually one or maybe few central directories. This centralised approach has the inherent drawback of a single point of failure. Further, centralised server(s) can also become a registration bottleneck in a highly dynamic environment where many resources join, leave, and change characteristics. Thus, it does not scale well within a large number of Grid nodes. P2P systems exhibit characteristics required to overcome the above-mentioned problems, improving scalability, performance and fault-tolerance.

In this paper, we exploit an existing P2P framework that is able to resolve multi-attribute and range queries using the K-Dimensional (K-D) [3] distributed tree structure. Whilst some solutions that have been proposed to tackle the problem of resolving multi-attribute and range P2P queries use Distributed Hash Tables (DHTs), these approaches suffer from a high overhead for maintaining multiple DHTs. In fact, the complexity for updating routing tables within a network of N peer nodes is $O(\log^2 N)$. In addition, DHT-based systems do not exhibit locality properties, i.e. resources location are determined by their hash value, not by their physical location. To overcome the aforementioned limitations, we use the Distributed K-D tree-based approach in [3] for efficient handling rich queries encompassing multiple attributes and range searches. The advantage of this is that rich queries are supported more efficiently, requiring O(log(N)) complexity for inserting and deleting new nodes, while locality of data is also preserved.

Load balancing in K-D based P2P overlays, will ensure that the complexity for resolving multi-attribute and range queries remains logarithmic with respect to the number of participating peers. We apply this algorithm to support rich queries encompassing services within a Grid environment, e.g. range queries on multiple-attributes of distributed resources.

The rest of the paper is organized as follows. In Section 2 we summarise related work. In Section 3 we outline how a K-D tree is created. In section 4 we present in detail the algorithm we devised for dynamic load balancing of a distributed K-D tree structure. Details of how our load balanced P2P system can be used for the discovery of Grid services are presented in Section 5. Finally, the last section concludes the paper and introduces our future research and implementation plans.

2. RELATED WORK

Several research groups have worked on *Distributed Hash Table (DHT)* P2P systems, tackling the problem of scalability, which is the main drawback of earlier P2P systems, such as *Kazaa* [4] and *Gnutella* [5]. Scalability in the context of P2P refers to the issue of how many messages are required to be forwarded within the network in order to locate a particular resource. In the worst case, N messages are needed to be sent to find a resource in a P2P network of N nodes, i.e. query every participating node. However, since the number of nodes is usually large, it is not efficient to deploy and use systems with an O(N) complexity, thus leading to the development of DHT-based systems, providing O(logN) complexity. A detailed discussion on DHT-based systems is provided in [6].

DHT-based P2P systems provide support for: a) single attribute exact-match queries, e.g. *Chord* [7] and *CAN* [8] or b) multiple attribute and range queries, e.g. *MAAN* [9] and *Mercury* [10]. However, although they naturally support single-attribute exact-match queries, it is non-trivial to find an efficient way to support multiple attribute and range queries within a single DHT. DHT based systems, such as *MAAN* and *Mercury*, provide methods for supporting multi-attribute and range queries, but require maintaining multiple DHTs, one per attribute. This leads to performance degradation and replication of data. Researchers in the P2P area consider as a key open issue the support for non-trivial search predicates like range queries, multi-attribute queries and operators other than equality, usually join and sum operators [6]. To address these issues, they started investigating how distributed tree data structures, common in legacy databases, can resolve complex queries in a P2P overlay network.

Distributed tree-based P2P systems supporting multi-attribute and range queries include Princeton University's *SkipIndex* [11], Stanford University's *MURK* [12], Cornell University's *BrushWood* [1] and HP's *NodeWiz* [2]. These systems partition the data space using the K-D tree data structure. However, although they provide a scalable solution for the efficient resolution of K-attribute and range queries, the entire index may need to be rebuilt if the data distribution changes significantly. For high-volume insertions, and dynamic placement of new data points in the data space, the aforementioned systems do not provide any guarantees on the cost of load balancing.

Our algorithm, reported below, obtains load-balancing within the K-D tree data-partitioning scheme. Furthermore, it exhibits robustness as it does not change the tree pointers while executing, in contrast with other load balancing approaches such as *BATON Trees* [13].

3. CREATION OF K-D TREES

We assume a P2P network with resources described as "data points" within a multi-dimensional data space. These can be located by querying peer nodes, each holding or "owning" pointers towards a disjoint subset of resources, altogether comprising the universal set of shared resources. An example of a shared resource is a music file identified by its three attributes, i.e. artist, title and genre. Owners are assigned within P2P members, i.e. PCs holding information on locating the music file. Unlike P2P popular protocols such as *Kazaa* or *Gnutella*, we assume that a single data point exists throughout the network, pointed by a single "owner" not necessarily coinciding with the peer holding it.

Our load balancing algorithm operates in a K-D Tree [15] structure produced by *SkipIndex* [11] that partitions the multi-dimensional data point space into disjoint *peer regions,* each assigned to a unique peer node. K-D structures can also be constructed using *MURK*, *BrushWood* and *NodeWiz*. These approaches are based on flooding for resolving location queries among regions, thus exhibiting higher than logarithmic routing complexity. We selected, however, *SkipIndex* as it provides logarithmic complexity by associating a one-dimensional key per region in order to obtain an absolute ordering of the regions. This key captures the hierarchical creation of regions. The keys are then used to store the leaf regions in a searchable *Skip Graph* [14], which supports insertion, deletion and lookup based on a one dimensional key.

SkipIndex structures were chosen not only for their logarithmic complexity, but also because they do not use hashing and therefore they preserve the logical integrity of the keyspace. This also enables location of data regions (K-dimensional data range query), extending location services for a single data point. A peer (leaf-region) searching for a target data region, can navigate through the *Skip Graph,* so that the distance to the target region (measured in terms of the number of hops when traversing regions), is halved in every region traversal. Note here that for scalability reasons each peer maintains a partial view of the K-D tree to aid query processing and to determine the direction from a region towards the destination point.

Figure 1 shows an example of how *SkipIndex* partitions the data-space region (Figure 1a) into its corresponding K-D tree (Figure 1b). The figure depicts a 2-dimensional tree, meaning that each data point has 2 attributes, *x* and *y*. Briefly, as new nodes join the network, they are assigned a data region by partitioning an existing region via its "median", giving the new node the location coordinates or "ownership" of "half" the data points within the data region being partitioned. The other half is still kept by its former "owner" node. Spitting is performed as follows: As the first peer node bootstraps the systems, it owns all data points. The tree is built up gradually, starting from the root, i.e. the first node; as new peers join-in, *SkipIndex* alternates among the *x* and *y* axes used to select the splitting planes [15].

The *SkipGraph* used to navigate between regions is also constructed by *SkipIndex* and is presented in Figure 1c.

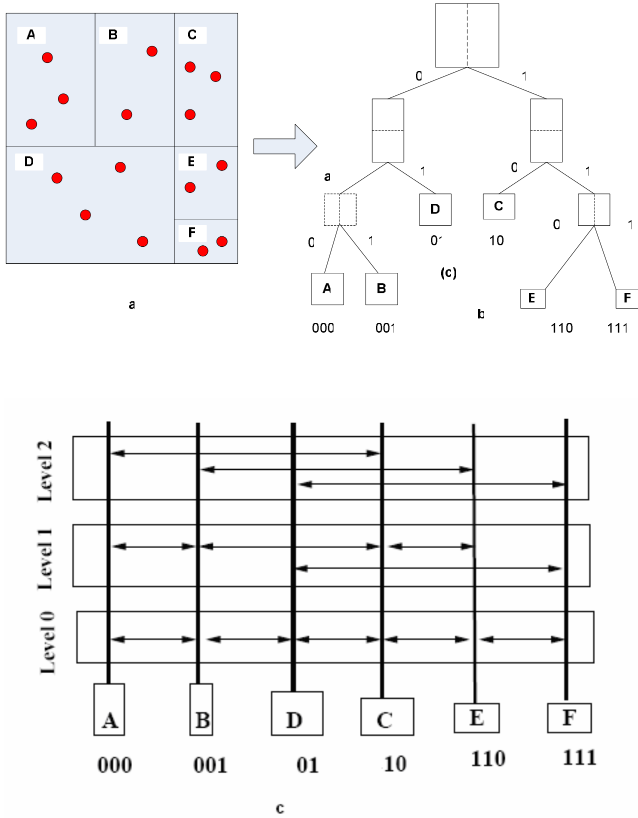

Figure 1. Partitioning of the data space and its corresponding Skip Graph

It consists of a number o linked lists, called *Skip Lists* [16]. Each element in a *Skip List* participates in several levels of linked lists. The lowest level list consists of all elements ordered by their keys. Each key that appears in the list at *Level i*, would also appear in the list at *Level i+1* with some probability p. This way, *Skip Graphs* extend *Skip Lists* for distributed environments by adding redundant connectivity and multiple handles into the *Skip Graph* data structure, thus improving resilience. At each level, a peer node stores pointers to its left and right neighbors. To locate a key, *SkipIndex* searches the highest level which might have just a few keys, dropping down to the more densely-populated lower levels, if needed, i.e. if we have not reached the target region. On average, there are O(logN) levels in the system, meaning that a search will traverse O(logN) nodes until it reaches its destination region.

As mentioned earlier, *SkipIndex* solves the scalability problem for resolving a P2P query by performing tree traversal in a distributed manner, without requiring any node to maintain the complete view of the index structure. This would result in an O(N) complexity when a new node enters the system, due to all N-1 nodes updating their connections to the new node. Instead, the innovation of this approach is that each node maintains a partial view of the whole K-D tree. The partial tree view or local tree view of a node comprises of the split histories of its local region and that of the regions maintained by its *Skip Graph* peers, where each split history provides information about the path from the tree root to a leaf region. Figure 2 illustrates the partial views of three index nodes. Note that in this figure, only the bottom-level *Skip List* is given. Of course, each node maintains additional links to other regions, according to the pointers in the higher level *Skip Lists* of the *Skip Graph* (see Figure 1c)

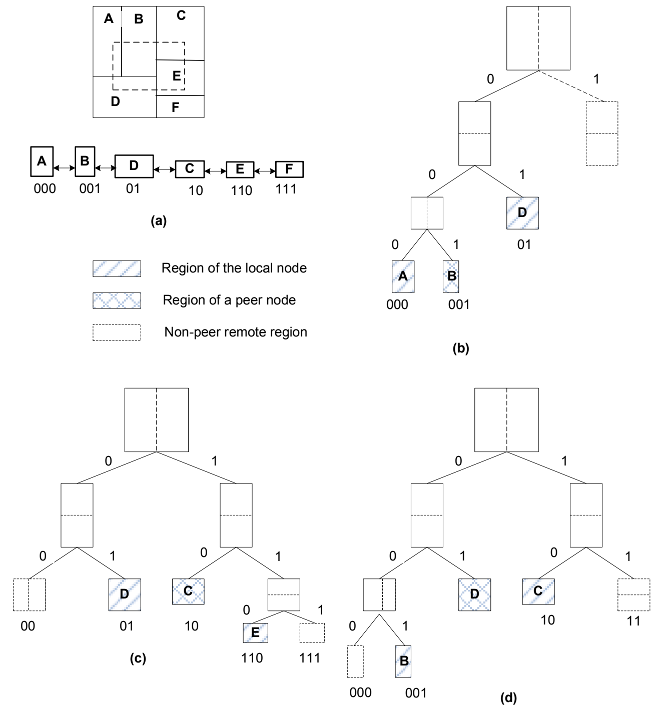

Figure 2. Partial views of trees among SkipIndex peers

The tree traversal for routing a query in the topology shown in Figure 2 does not descend the tree sequentially, but rather "jumps" into sub-trees. The maximum number of hops to reach a leaf region that belongs to the target area, does not depend on the number of peers, which would result in an O(N) complexity. Instead, the routing complexity is O(logN), as the routing process forwards the message using links in the *Skip Graph*, and at each step, the distance to the set of nodes representing the target region is halved, resulting in the O(logN) complexity. Remember that this is also exhibited by the *Chord* DHT routing protocol which, however, does not handle multi-dimensional and range queries.

In the following section, we describe how our algorithm achieves load-balancing (leaf regions with equal number of data points) within a *SkipIndex* created K-D tree.

4. A NEW ALGORITHM FOR DYNAMIC LOAD-BALANCING OF K-D TREES

Our framework improves a number of existing K-D tree-based P2P systems with a new algorithm for dynamic load-balancing the distributed K-D data structure used to organize the P2P overlay. Load balancing is necessary, since O(logN) routing complexity is achieved only when the tree remains load-balanced, e.g. upon a new node joining the network an existing data region is partitioned in two sub-regions, each containing the same number of data points.

As discussed in the previous section of the paper, we will use *SkipIndex* as an example of a K-D tree based P2P system and we will show how our algorithm can help maintaining the resources (data points) load-balanced among the participating nodes. Note that our algorithm is not specifically implemented for the *SkipIndex* framework. Rather, it provides a general idea that can be implemented and instantiated within different P2P systems that use any distributed K-D structure, other than the one used within the *SkipIndex* framework.

Within a K-D enabled P2P system, to achieve load-balancing during system's operation, it is necessary to change the ownership of a data set to another node or a set of nodes. This is required when the set of data points, i.e. the shared resources or total load of the system can dynamically change, placing new data points in the K-dimensional data space or removing data points. For this purpose, our algorithm redistributes data points in a recursive manner, reacting to changes. Using *SkipIndex* as an example, to decide which region or region set should be re-assigned a load, we have extended the *Skip Graph* that *SkipIndex* uses with an additional key, which maintains information about the load of other nodes (number of data points), based on the split history of that node. This is analogous to the procedure followed by *SkipIndex* to maintain the split history, as described in section 2 of the paper.

The following figure presents an example of how the "load" history is kept within the leaf nodes (i.e. the peers of the P2P overlay) of the K-D tree, according to their split history. The key representing the load history of a node is a vector of log(N) values, N being the total number of peers in the network (regions in the K-D data space). The first value of the node load vector represents the number of data points kept by the node. Note here that upon construction of the K-D tree, this value is the "ideal" or "expected" value of the load of the node. This is because the algorithm to construct the K-D tree [17], assigns data points to the participating peers so that the load among peers is equally distributed. The remaining load vector values represent the load of the parent node when splitting occurred when inserting the current node within the K-D data space.

$$\vec{A}_{load} = \{30, 60, 120, 240\}$$
$$\vec{B}_{load} = \{30, 60, 120, 240\}$$
$$\vec{D}_{load} = \{60, 120, 240\}$$
$$\vec{C}_{load} = \{60, 120, 240\}$$
$$\vec{E}_{load} = \{30, 60, 120, 240\}$$
$$\vec{F}_{load} = \{30, 60, 120, 240\}$$

Figure 3. Keeping the load history of peers

In Figure 3, we show the load history of all nodes *A, B, C, D, E* and *F* within the K-D tree of Figure 1. Node *A* with key 000, holds a vector consisting of

$$\lceil \log_2(N) \rceil + 1 = \lceil \log_2 6 \rceil + 1 = 4$$

elements. The first item of the vector holds *A*'s local load, 30 points in this example, while other items hold the load history of the parents of *A*, at the time when splitting occurred. Note that the value of 30 points is the ideal or expected load value of node *A*; this was achieved by *SkipIndex* while constructing the K-D tree, assigning peer nodes (i.e. the leaf nodes of the tree) an equal number of data points to keep locally.

The pseudo-code shown in Figure 4 describes our proposed algorithm *LOAD_BALANCE* executed when a load change Dl_{oad} is observed within an individual node[1] *X*, The algorithm is triggered when a relative load change exceeds a *THRESHOLD* parameter, set by the administrator. The relative load change is the percentage of the actual load change by the ideal or expected, as we discussed above, load of the node. It is given by the following formula:

$$relative_load_change = \frac{NODE.D_{LOAD}}{NODE.Expected_{LOAD}}$$

The above formula is used to decide whether a load change can be tolerated by a node, without requiring load redistribution in order to preserve the load balancing property of the K-D tree. In other words, when the load change Dl_{oad} is relatively small against the expected load of the node, new data points are added to the node without violating the requirement for an overall load-balanced tree.

```
LOAD_BALANCE(X, Dl_oad ,THRESHOLD)
1.  if Dl_oad /X.LOAD  <= THRESHOLD
2.     then return
3.     else
4.          N_SLPIT=FIND_LOAD_SPLIT_NODE(X, Dl_oad ,
                                           THRESHOLD)
5.          LOAD_BALANCE(X, Dl_oad /2,THRESHOLD)
6.          Y = FIND_KEY(N_SPLIT,X)
7.          LOAD_BALANCE(Y, Dl_oad /2,THRESHLOD)
```

Figure 4. Load balancing algorithm LOAD_BALANCE

Unlike other reported work on load balancing [16], our algorithm *(LOAD_BALANCE)* does not treat the load change Dl_{oad} as a whole set of data points that must be assigned to another leaf of the K-D tree. Rather, in case when the current node is not able to tolerate the load change by itself, half of the load change is recursively being assigned to another leaf node of the tree, node *Y* in Line 6 of the pseudo-code in Figure 4. The other half of the load change is handled recursively by the current node. The way to determine the remote node *Y* is given with the procedures *FIND_SPLIT_NODE* and *FIND_KEY* within *LOAD_BALANCE*. We will explain now how these two methods operate, giving their pseudo-code in Figure 5.

[1] The LOAD_BALANCE algorithm must be followed by multiple nodes upon changes. There may be cases with two nodes executing the algorithm resulting into oscillations. This stability issue is not addressed in this paper.

```
FIND_SPLIT_NODE(X, Dl_oad ,THRESHOLD)
1.  I =1; N = X.load_history_vector(I);
    MAX =   X.load_history_size;

2.   while (Dl_oad /N.LOAD  <= THRESHOLD AND I<MAX)
3.           do I = I+1; N = X.load_history_vector(I);
4.   return N;


FIND_KEY(N_SPLIT,X )
1.   if X.key < N_SPLIT .key  // Y is located to the right of X. Check
the Skip Graph to find the most right node of the subtree with root
N_SPLIT
2.      then
return TRAVERSE_SKIP_GRAPH_TO_KEY (N_SPLIT .key +1)
3.      else // Y is located to the left of X. Check the Skip Graph to
find the most left node of the subtree with root N_SPLIT
 return TRAVERSE_SKIP_GRAPH_TO_KEY (N_SPLIT .key -1)
```

Figure 5. Methods FIND_SPLIT_NODE and FIND_KEY used to discover remote node *Y* in LOAD_BALANCE

FIND_SPLIT_NODE reads the load history vector of current node *X* to find out which parent is able to tolerate the load change. The complexity of this method is O(logN), where logN is the size of the load history vector, as discussed earlier. When the node N_{SLPIT} in Line 4 in the pseudo-code in Figure 4 is discovered by *FIND_SPLIT_NODE* (i.e. the intermediate tree node that is able to tolerate the load change Dl_{oad}), then we must locate the leaf node *Y*, which is a leaf of the subtree with root N_{SLPIT}. This is accomplished by the method *FIND_KEY* that checks the Skip Graph to find the node *Y* based on *Y*'s Skip Graph key. Like *FIND_SPLIT_NODE*, the complexity of *FIND_KEY* is O(logN), as this is the routing complexity to find a particular key within the Skip Graph. The Lemma (at the end of the paper) proves that the overall complexity of our load balancing algorithm is $O(\log^2 N)$, in a network of N leaf nodes (i.e. N peers) in the K-D data space.

We will now describe with an example how the recursive algorithm operates when instantiated with the K-D tree shown in Figure 6.

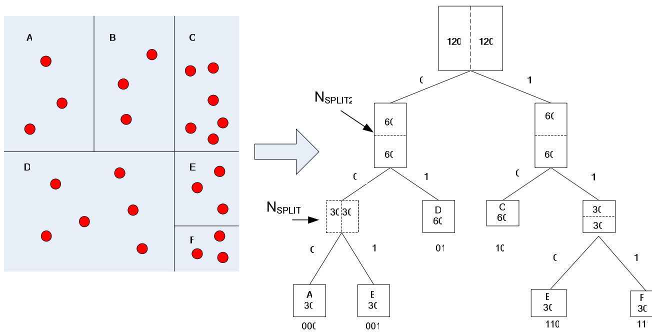

Figure 6. Operation of the LOAD_BALANCE algorithm

Initially, node *A* holds 30 data points (each circle in the diagram represents a small region of 10 data points within it). This is the ideal or expected load that node *A* should have so that the whole tree is kept load balanced. Assume now that during system's operation the load of *A* changes, by assigning more data points to his region. In a P2P context, this is analogous of adding more resources within a node to share within the P2P overlay. There are several choices that can be made when *A* detects the load change Dl_{oad}; this choice depends on the value of *THRESHOLD*, in this example chosen to be 20%. Our algorithm determines the choice that should be made as follows:

Case 1: ***Dl_{oad} /A.LOAD ≤ THRESHOLD***

There is no need to re-adjust load to other regions and the K-D tree still remains load-balanced. Note here that we do not change the expected load value of the current node, even if the load in the node changes upon a new load offered to the node; node *A* in our example will keep the value of 30. Thus *THRESHOLD* will be eventually violated, even when the load in a region (i.e. the load of a node) increases slowly over time.

When, after several small changes, the threshold is violated, our algorithm will send a portion of the load change to other region(s), so that the whole tree remains load balanced.

As we discussed before, in Figure 3, the numbers which we use to calculate the relative load change, refer to the "expected load value" per every node, i.e. 30 is the expected load value of *A*. This means that if *A* adds 5 and then another 5 points, in the first case, the relative load change is 5/30 = 16.6% < *THRESHOLD* (20%). This means that node *A* is able to keep for himself the five (5) new data points. But, next, when we try to add to *A* another 5 points, then the relative load change (5+5)/30 = 32.3% exceeds the 20% *THRESHOLD*; the resulting redistribution of the load is described below in Case 2.

Case 2: ***Dl_{oad} /A.LOAD > THRESHOLD***

The change of load demands redistribution of the load change to another region or set of regions. To determine which region(s) should be given the load change, we will consult the "load" history of the node using the procedure *FIND_LOAD_SPLIT_NODE* to find out which is the root (parent of the current node A) of the subtree that is able to hold the load change.

In our example, assume that *A* is given 10 more data points. Since the parent of *A* (as found by the load history) has a total load of 60 points, the parent load change (10/60) is below the threshold of 20%. This means that *A* knows that the load splitting node is his parent node N_{SPLIT1} with key 00, thus half of the load change (5 points) must be assigned to node *B*, which is the most left node of the subtree with root *A*'s parent node N_{SPLIT1}. The other 5 points will be added to *A*'s region, resulting in a parent's load of 70 points. Both *A* and *B* are able to handle the 5 points load change (5/30 < 20%) so their recursion will terminate in the first call of *LOAD_BALANCE* shown in Lines 5 and 7 in the pseudo-code in Figure 4.

In another instance of the aforementioned network, Dl_{oad} has a value of 20. In this case, the load splitting node is discovered by *A* and is the node N_{SPLIT2} in Figure 6. To distribute the load change, *A* will request that the load

change is divided in two equal parts (each of a size of 10 points), one given to the most left child *D* of the node N_{SPLIT2}, while the other will be re-examined by *A*. Thus, the same procedure will be executed recursively by node *D*, with input Dl_{oad} of 10 points. As *D* is a leaf node, *D* will keep all 10 points for himself, *as* 10/*D.LOAD* (10/60) is below the threshold of 20% for node *D*. As discussed earlier, the other 10 points will be split between *A* and *B*, *A* executing recursively the procedure *LOAD_BALANCE* with an input of 10 points of load change.

Note that in the example discussed above, the only information that *A* needs in order to distribute half of the load change is the location of node *D*, i.e. the node that must take over half of the load change for further distribution. As node *A* does not have a full view of the whole K-D tree, it must dynamically discover the location of node *D*. This is accomplished by the procedures *FIND_SPLIT_NODE* and *FIND_KEY* that will locate the splitting node and the leaf node *D* respectively. Since *A* knows from his load history that the root of the tree that can hold the load change is node N_{SPLIT2} in Figure 6, this means that the node that can be assigned the half of load change is the most left node of the right sub-tree with root N_{SPLIT2}. To find out this node, the only thing that *A* must do is to find the node with key with a minimum value of 01 (N_{SPLIT2}..*key* + 1) which is the key of the leaf node *D*. Should the threshold for *D* be exceeded, the latter would execute *LOAD_BALANCE* recursively.

If the load change can not be tolerated even by the root of the tree, the procedure will be executed a number of times, in a sequential manner: We split the load change in smaller pieces, each with a value of *THRESHOLD*ROOT.LOAD*, where *ROOT.LOAD* is the current load of the root, thus each piece can be marginally accommodated. Otherwise, it could be possible to fall into oscillations, with the left sub-tree of the whole K-D tree assigning data points to the right sub-tree and vice versa.

In summary, each time the load change to be handled is *THRESHOLD*ROOT.LOAD*, incrementally adding points to the whole tree, until all points Dl_{oad} are finally distributed. The number of times *LOAD_BALANCE* will be executed is given by:

$$\left\lceil \log_{1+THRESHOLD} \frac{ROOT.LOAD + D_{LOAD}}{ROOT.LOAD} \right\rceil$$

The complexity of the *LOAD_BALANCE* algorithm is $O(\log^2 N)$ (see the Lemma) for a P2P network of N peer nodes. All inner operations of the algorithm, finding the load splitting node, finding the most left of the right subtree and the most right of the left subtree nodes, require an O(logN) number of steps to query the load history vector and the *Skip Graph* respectively.

Note that in all cases above, when node(s) receive a request to adjust their load, they must follow, along with their O(logN) *Skip Graph* peers, a procedure to update accordingly their load-history vectors. This is done in order to maintain an updated load history within the overall K-D tree. Furthermore, apart from the load history vectors that reflect the ideal load distribution of the network, peers maintain additional information on the current value of their load. For example, if node *A* adds 5 points to himself, although his load history vector will retain the value of 30, node *A* will know that his current load is 35 (30 is his expected value) and also, the current values of his parents are 65, 125 and 245 respectively. Again, the expected values of *A*'s parents will retain their values {60,120,240}. Finally, when the K-D tree becomes perfectly load balanced, the load history vector holding the expected values will be updated accordingly.

We add here that all insert (join) and remove (leave) operations of peers within the P2P network are not handled by our algorithm, which is used solely to maintain the K-D tree structure load-balanced. The algorithm to handle join/leave operations of peers (not data within peers) is the *SkipIndex* K-D tree algorithm [11], which deals with inserting and removing nodes with an O(logN) complexity.

Last, note that a query is still successfully executed, while the load balancing algorithm redistributes load changes. In fact, unlike other load balancing approaches, e.g. *BATON Tree* [13], that move or merge leaves, our algorithm is robust, i.e. we do not reconstruct the tree when re-distributing the load upon requests for load changes. We keep exactly the same tree and its pointers so the query will return all the correct results after O(logN*logN), the time used by our algorithm to re-distribute the load. In-between, the system will return the "old" set of resources, but still will not fail to execute, which is the case of a P2P system handling dynamically new resources.

5. CASE STUDY: IMPLEMENTING A DISTRIBUTED GRID INFORMATION SERVICE

In a Grid environment, Information Lookup Services are needed to resolve multi-attribute and range queries, more complex than finding a service based on a given name. For instance, a user would like to find out which services exhibit certain characteristics, such as availability, price, etc. We propose a load balanced K-D based P2P platform to enhance existing Grid middleware to enable scalable, decentralized discovery of shared Grid resources. This proposal is an alternative to centralized services, such as the Universal Description, Discovery and Integration protocol - UDDI [17], exhibiting fault tolerance due to its fully distributed nature. In this section, we will illustrate our ideas on a P2P-based Grid Information Service, independent of any underlying Grid platform implementation using abstractions of Grid shared resources, rather than actual implementation information models and protocols.

To support an implementation independent Grid Information Service, we must first determine how entities to be discovered can be formally represented, i.e. what are their functional and non functional abstractions on which their selection can be based. To that end, we specified a suitable information model, necessary to define and know a-priori the type of attributes, on which queries for Grid resources/services are based. In simple cases, such as the case of file-sharing, this may not be necessary, as the only resource shared among nodes are data files, all identified by their names. Instead in a Grid environment, models of complex shared resources & services are required. Figure 7 presents the UML model of shared Grid resources/services within our multi-service agent-assisted Grid framework [18].

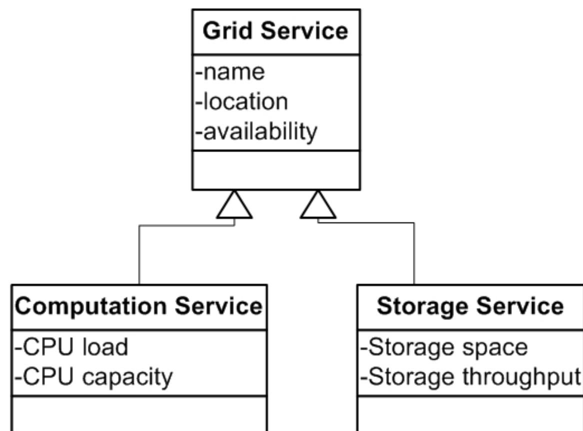

Figure 7. Example Grid Services Information Model

In the above information model, a Grid Service is identified by its name, its location and its availability, thus by three attributes. Computation and Storage services inherit these attributes and include additional attributes for their description. Note here that in this Information Model, we tried to keep the cardinality of attributes as small as possible, taking into account the fact that it the complexity to implement and maintain any P2P system increases as the number of resource attributes increases. It is important to note at this point that the information model we introduced is used at the architectural layer of the P2P middleware. Higher layers, such Grid service layer can use more complex structures for representing information, such as WSDL [18]. However, when higher level components need to use the P2P Grid Information Service for discovery, the entities to be discovered must conform to the information model understood by the P2P middleware. Upper layers need to include mechanisms to translate their information model to the low-level model shown in Figure 7.

Based on the information model we have designed, we will construct analogous K-D data structures, according to user requirements for locating Grid resources. In the simplest case, we will only deploy a single K-D tree with 3 dimensions (K=3), namely the name, the location and the availability of a Grid Service. These three attributes will be used to define P2P queries on Grid services as 3-attribute and range queries. We use the the *SkipIndex* [11] framework to partition the 3-D tree data space in a number of regions, discussed in section 2. The load balancing property will be maintained dynamically by using our algorithm *LOAD_BALANCE* for redistributing the load of the 3-D tree, i.e. the number of shared Grid services. In a more complex case, we can construct two 5-D trees representing the Computation and Storage Services respectively. In both trees, the *LOAD_BALANCE* algorithm will be operated to keep them load-balanced. This way in a network of N peer Grid sites, P2P queries involving 5-attribute range queries on Computation and Storage services are resolved with an O(logN) routing complexity, while the complexity to keep the trees load-balanced is $O(\log^2 N)$.

6. CONCLUSIONS AND FUTURE WORK

This paper presented a novel algorithm for dynamic load-balancing the distributed K-D tree structure used for organising the shared information within a P2P network. Load balancing is very important to achieve, as this ensures that the complexity for resolving multi-attribute and range queries remains logarithmic in respect to the number of participating peers. We presented in detail how the load-balancing algorithm operates. Evaluation of its performance within a simulation environment is within our current activities. We are implementing a simulator to assess our P2P system in comparison to other multi-attribute and range queries supporting systems, such as *MURK*, *MAAN* and *Mercury*. Our experiments focus on dynamic environments, with load varying at run-time within participating nodes, which continuously join and/or leave the network.

We illustrated how our P2P framework can be used to locate resources within a GRID environment. We plan to provide a full integration implementation between our P2P framework and a Grid platform, such as *gLite* [19] or *GRIA* [20], as a means to implement a fully distributed P2P-based Grid Information Service. This implementation will be tested with a number of applications over a shared Grid infrastructure. We believe that this work will support a class of Grid applications, exploiting the efficiency of a P2P system for locating in a scalable and robust way distributed resources/services in a dynamic, multi-service Grid environment.

7. ACKNOWLEDGMENTS

The work presented in this paper was partially supported by the EC project ARGUGRID-IST-035200.

8. REFERENCES

[1] S. Basu, S. Banerjee, P. Sharma, S.J Lee: NodeWiz: Peer-to-peer Resource Discovery for Grids, in Proc. of the IEEE International Symposium on Cluster Computing and the Grid, (CCGrid 2005), pp213-220, May 2005.

[2] Chi Zhang Arvind Krishnamurthy Randolph Y. Wang. Brushwood: Distributed Trees in Peer-to-Peer Systems. In Proc. 4th International Workshop of Peer to Peer Systems, Ithaca, NY, Feb. 2000.

[3] J. L. Bentley: Multidimensional binary search trees used for associative searching. Commun. ACM, 18(9), 1975.

[4] Kazaa. Available from http://www.kazaa.com

[5] Gnutella. Available from http://www.gnutella.com

[6] Risson, J. and Moors, T. Survey of research towards robust peer-to-peer networks: Search methods. In Computer Networks, Volume 50, Issue 17, 5 December 2006

[7] Stoica I., Morris R., Karger D., Frans Kaashoek M., Balakrishnan H.: Chord: A scal-able peer-to-peer lookup service for Internet applications, In Proc. ACM SIGCOM, San Diego, 2001.

[8] S. Ratnasamy, P. Francis, M. Handley, R.M. Karp S. Shenker: A Scalable Content-Addressable Network, in Proc. ACM SIGCOMM 2001 Conf. on Applications, Tech-

nologies, Architectures, and Protocols for Computer Communication, pp. 161-172, 2001.

[9] M. Cai, M. Frank, J. Chen, P. Szekely: MAAN: A multi-attribute addressable network for grid information services, In the 4th Inernational Workshop on Grid Computing, 2003.

[10] A.R. Bharambe, M. Agrawal S. Seshan: Mercury: Supporting Scalable Multi-Attribute Range Queries, in Proc. ACM SIGCOMM 2004 Conference on Applica-tions, Technologies, Architectures, and Protocols for Computer Communication, pp. 353-366, 2004.

[11] C. Zhang, A. Krishnamurthy, R. Y. Wang: SkipIndex: Towards a scalable peer to-peer index service for high dimensional data. Technical Report TR-703-04, Prince-ton Univ. CS, 2004.

[12] P. Ganesan, B. Yang H. Garcia-Molina: One Torus to Rule them All: Multidimen-sional Queries in P2P Systems. In Proceedings of the 7th International Workshop on the Web and Databases (WebDB '04).

[13] Jagadish, H. V., Ooi, B. C., and Vu, Q. H. 2005. BATON: a balanced tree structure for peer-to-peer networks. In Proceedings of the 31st international Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 02, 2005.

[14] J. Aspnes, G. Shah: Skip Graphs. In Proceedings of Symposium on Discrete Algorithms, 2003.

[15] Friedman, J.H., Bentley, J.L., and Finkel, R.A.. An algorithm for finding best matches in logarithmic expected time. *ACMTransactions on Mathematical Software*, 3(3):209–226, 1977.

[16] Pugh, William. Skip Lists: A Probabilistic Alternative to Balanced Trees. In Proceedings of Workshop on Algorithms and Data Structures, pp. 437-449", 1989.

[17] The Universal Description, Discovery and Integration (UDDI) protocol. Available from http://www.uddi.org

[18] Web Services Description Language (WSDL). Available from http://www.w3.org/TR/wsdl

[19] gLite. Available from http://glite.web.cern.ch/glite/

[20] GRIA. Available from http://www.gria.org

LEMMA

The complexity of the LOAD_BALANCE algorithm is $O(\log^2 N)$.

Let us denote T(H) the complexity of the load balancing algorithm when the height of the K-D tree is H, equal to $\lceil \log N \rceil$. Upon each recursive call of the procedure, we will find a load splitting node a level down each time the recursion occurs, e.g. if the first call finds the root of the tree (level 0 node) as the load splitting node, then the second call will find (in the worst case) the level 1 node as the splitting node and so on. Then

$$T(H) \leq T(H-1) + O(H) = O(H^2), \text{ thus } T(N) \text{ is } O(\log^2 N)$$