A LOAD BALANCING ALGORITHM BASED ON REPLICATION AND MOVEMENT OF DATA ITEMS FOR DYNAMIC STRUCTURED P2P SYSTEMS

Narjes Soltani and Mohsen Sharifi

Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

ABSTRACT

Load balancing is one of the main challenges of every structured peer-to-peer (P2P) system that uses distributed hash tables to map and distribute data items (objects) onto the nodes of the system. In a typical P2P system with N nodes, the use of random hash functions for distributing keys among peer nodes can lead to O(log N) imbalance. Most existing load balancing algorithms for structured P2P systems are not adaptable to objects' variant loads in different system conditions, assume uniform distribution of objects in the system, and often ignore node heterogeneity. In this paper we propose a load balancing algorithm that considers the above issues by applying node movement and replication mechanisms while load balancing. Given the high overhead of replication, we postpone this mechanism as much as possible, but we use it when necessary. Simulation results show that our algorithm is able to balance the load within 85% of the optimal value.

KEYWORDS

Structured P2P Systems, Load Balancing, Node Movement, Replication

1. INTRODUCTION

Distribution of objects among nodes in most structured Peer-to-Peer (P2P) systems is done through Distributed Hash Table (DHT) mechanism that use consistent hashing to map objects onto nodes [1,2,3,4]. Using this mechanism, a unique identifier is associated with each data item (object) and each node in the system. This is simply shown in Figure 1. The identifier space is partitioned among the nodes that form the P2P system and each node is responsible for storing all data items that are mapped to an identifier in its portion of the space.

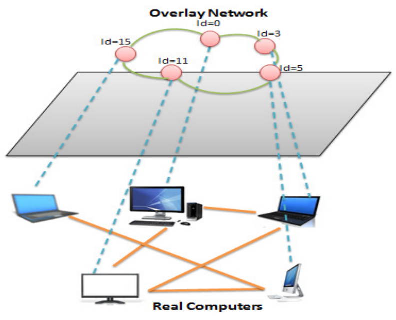


Figure 1. A schematic view of a structured p2p system

If node identifiers are chosen at random (as in [1,2,3,4]), a random choice of item identifiers results in *O(log N)* imbalance factor in the number of items stored at a node, where *N* is the total number of nodes in the system. Furthermore, the imbalance may be due to the non-uniform distribution of objects in the identifier space as well as high degree of heterogeneity in object loads and node capacities, memories, and bandwidths.

Most existing load balancing algorithms in structured P2P systems, do not consider system dynamicity, nodes and objects heterogeneities, links latencies between nodes, and the popularity level of moved data items (objects) [5,6,7]. Furthermore, they are totally ignorant as to the causes of nodes overloading.

In this paper we present a new load balancing algorithm which is based on our previous work [8] and differentiates two cases for node overloading: 1) when only one data item on a node is highly popular, and 2) when more than one item is popular or a high number of data items are mapped to a node but none of the items is highly popular. Our algorithm differentiates these two cases and uses two mechanisms namely object replication and node movement to balance the load between nodes considering the popularities of items. To avoid the overheads of node replication, our algorithm uses node movement for the purpose of load balancing in most cases. It only uses the replication mechanism when it detects that an alone node is not able to handle its only popular data item. The algorithm uses capable nodes to handle load balancing. Also in order to consider system's varying loads at different times, it introduces a new notion called *valid boundary* and balances the load considering the following parameters:

1. Non-uniform distribution of data items
2. System heterogeneity
3. The different popularity levels of data items
4. Link latency between nodes

The rest of the paper is organized as follows. Section 2 presents related works. Section 3 explains in more detail and formulates the load balancing problem. Section 4 presents our load balancing algorithm. Section 5 explains how system directories are stored in more capable nodes to help in system load balancing. Section 6 shows the performance of our algorithm through simulation. Section 7 concludes the paper and introduces some future directions.

2. RELATED WORK

Generally load balancing protocols are divided into two main groups in structured P2P systems. The first group is based on uniform distribution of data items (objects) in identifier space and the second group has no such assumption [9]. Suppose that there are *N* nodes in the system, load balancing is achieved in the first group if the fraction of address space covered by each node is *O(1/N)*. Most algorithms have used the notion of virtual servers, first introduced in [1] to achieve this goal. A virtual server is similar to a single peer to the underlying Distributed Hash Table (DHT) and has its own routing table and successors list, but each physical node can take the responsibility of more than one virtual server.

There are two main advantages in using virtual servers. The first advantage is that nodes can own noncontiguous portions of identifier space when they have multiple virtual servers. The second advantage is that virtual servers have the ability to move from any node to any other node in the system; this can easily be done by using a leave followed by a join in the underlying DHT which is supported by all DHTs.

Chord [1] allows each physical node to host *O(log N)* virtual servers so that each node gets a constant number of items with high probability. But this solution has some drawbacks; for example it assumes uniform load distribution on nodes, assumes all nodes have the same capacities, and it uses a constant number of virtual servers for every node, a choice which is only effective in homogenous systems. Also Chord load balancing algorithm is nonreactive and it tries to balance the load only when new nodes join the system. In other words, it has no provision to redistribute the load, making it unsuitable for dynamic structured P2P systems.

CFS [10] accounts for nodes heterogeneities by allocating to each node some number of virtual servers proportional to the node capacity. In addition, CFS proposes a simple solution to shed the load from an overloaded node by having the overloaded node remove some of its virtual servers. However this scheme may result in thrashing as removing some virtual servers from an overloaded node may result in another node becoming overloaded. The reverse operation is done in the case of node underloading by creating some new virtual servers for the underloaded node, but it has its own problems again. A node with only some limited number of virtual servers may have no accurate estimation of the costs of creating new virtual servers. Also when the whole system is in an underloading status, it is quite probable that every node creates its maximum allowed number of virtual servers resulting in a huge increment in the sizes of routing tables and the search time. The main advantage of CFS is that it is completely decentralized.

Rao et al. [5] have proposed three different mechanisms to balance the load using virtual servers, yet their mechanisms are static and ignore data items popularities.

Using virtual servers in any algorithm leads to some common disadvantages. The first is that it leads to churn increase. It means that when a physical node wants to join the system, it should do the join operation for all of its virtual servers and when it wants to leave the system, it should remove all of its virtual servers. Because joining and leaving of objects from structured P2P systems impose some overhead, using virtual servers causes this overhead to be multiplied. Also in most structured P2P systems, searching is guaranteed to be done in *O(log N)* steps, but when using virtual servers this value changes to *O(log M)* where M is the total number of virtual servers in the whole system. Another disadvantage of virtual servers is that they increase the size of routing tables. Considering above problems with virtual servers, we do not use virtual servers in our proposed algorithm.

Protocols which do not assume uniform object distribution use two different mechanisms to achieve load balance, namely object movement [9] and node movement [11]. Movement of objects breaks the DHT assigned addresses of objects to nodes making it hard to find objects further on. Its application is thus limited to situations where the latter is not an issue, e.g. when objects correspond to programs that do not have to be found because they transmit their results automatically.

Moving nodes by letting them to choose their own addresses arbitrarily increases the threat of Byzantine attack that can prevent some items from being found in the network. This is done by simply choosing the node's address to be the address of the item; the node then becomes responsible for storing the item and can refuse to do so. Moving nodes preserve the DHT search mechanism.

Apart from the above classification, replicating data items is another way to achieve load balance. Akbariani et al. [12] have proposed a replication mechanism on Chord. They have used multiple hash functions to produce several key identifiers from a single key. Their approach has one main drawback that is they have defined no way to determine the number of hash functions for replicating a data item. Xia et al. [13] have discussed this problem and have proposed a solution for it. A major limitation about their solution is ignoring nodes heterogeneity while replicating, i.e. replication is done blindly on some nodes and without considering their ability.

CAN [2] has proposed two approaches for data replication. The first approach uses multiple hash functions, but this is done statically and without considering system varying load patterns at different times. In the second approach, a node that finds it is being overloaded by requests for a particular object can replicate the object at each of its neighbouring nodes. The main problem with this approach is that system cannot control objects replication on nodes with more capabilities.

In Pastry [3] replicas of an object are stored on the k Pastry nodes with keys numerically closest to the object's key. The problem with this approach is again not having the ability to replicate objects on more capable nodes.

Although some other replication mechanisms such as the one presented in [14] have been proposed for structured P2P systems, but they are mostly concerned with placing object replicas to maximize system availability with no concern for balancing the load of objects on the nodes of the system. Also most of the proposed replication algorithms use to replicate data items from the early operation of the system in specified number of nodes which seems not to be necessary especially for unpopular data items, as it leads to wasting of system resources and increase system complexity.

3. DEFINITIONS AND PROBLEM FORMULATION

In this section we explain the primary definitions in context of load balancing and also present our formulas.

3.1. Primary Concepts

In most structured P2P systems, the load of each node is defined as the number of items that are stored in that node [5,6,7,15]. On the other hand, there may exist high popularity items that make a node overloaded, although it stores some limited number of data items. Therefore, by taking into account the number of entered requests for a node's data items, we can define a node's load as the average number of bytes that are transferred by that node in each unit of time. By the same token, we can define the node capacity as the maximum number of bytes that node can transfer per time unit.

When a node intends to join the system, it is given a unique key using a hash function we call the *FirstHash* hereafter in this paper. For the purpose of load balancing, a set of load directories, each called *LoadDir*, is designed in the system to which nodes send their loads, capacities, locations, and downtimes periodically. A node's downtime is defined as the summation of continuous times it is not accessible by other nodes in the system in defined intervals namely *t* that can be due to links failure or high traffic. We use the successor and predecessor nodes of each node *n* to periodically ping node *n* and to record the current system time *tt* somewhere in their local memories in case they find node *n* inaccessible. If after a specified timeout interval they again find node *n* as their successor (or predecessor), they subtract *tt* from the current system time and report this value to node *n*. The downtime of node *n* is periodically estimated by summing the reported values in the related *t* interval. We store locations of nodes to consider locality during load balancing process and thus reduce the time it takes to balance load.

Later we explain the way these directories are stored in nodes that are more capable in terms of bandwidth, uptime and memory in comparison with other nodes. Each node should be aware of a node that stores its related directory. Specifying directory store nodes however is done dynamically by considering different system states. So we define some constant nodes as pointer nodes in which the identifier of directory store nodes are saved.

We use the approach proposed in [16] to prevent Byzantine attacks and to specify pointer nodes. Each node connects to a central authority once, i.e. the first time it joins the system and obtains a pointer identifier; we denote this pointer with *PointerNo* that specifies to which *LoadDir* the node should send its information. In other words, each *PointerNo* specifies a pointer node and each pointer node specifies the identifier of the node that stores the related *LoadDir*. The node whose identifier is equal to or follows a *PointerNo* is in fact the related pointer node for this *PointerNo*. The number of distinct pointer identifiers is limited and determines the number of load directories in the system.

Using the explained mechanism, a node cannot dishonestly report its information to take responsibility of some specified items and then refuse to respond to those items' requests (Byzantine attack). The reverse case happens when nodes report their information falsely to prevent movement of some items to them. To stop the occurrences of such cases, incentive mechanisms such as the ones presented in [17] can be applied. We simply suppose that nodes report their locations honestly, but more secure mechanisms like the one proposed in [18] can be applied.

The central authority periodically sends the pointer numbers to the related pointer nodes so that each directory can get aware of other directories.

In our algorithm we group the nodes based on their load directories. In other words, nodes that send their information to the same *LoadDir*, form a group with each other. The use of these groups is explained in the following part.

3.2. Load and Cost Definition

Our load balancing algorithm tries to minimize the load imbalance factor in the system while minimizing load movement. Also we consider some other important factors which are related to the destination of the load transfer. In our algorithm, the calculation of the cost of transferring load to a destination node is based on destination load, downtime and also its proximity to the overloaded node. By considering proximity, we want to show the importance of links' latencies in the final cost. The final goal is to increase the number of successfully routed queries in the system. Since there is no global information in P2P systems, we do not claim to select the best node in the system to move the load to it, but our algorithm does this in the defined groups. In other words, by using the following formulas, we want to select nodes in the related group that impose the minimum cost. So when we want to move some load from a node i to a node j the destination cost is formulated as below:

$$DestinationCost=w_1 * Load\_status_j + w_2 *(loc_i - loc_j)/distance_{max} + w_3 *(downtime_j/t) \quad (1)$$

$$Load\_status_j = (cap_{max} - cap_j)/ cap_{max} + load_j / cap_j \quad (2)$$

In (1), *cap* and *loc* denote the capacity and location of a node respectively. To normalize the location parameter in (1), we divide the result of subtracting locations by $distance_{max}$ that stands for the distance between *i* and the farthest node in the related group. By downtime, we mean the summation of continuous times the node is not accessible by other nodes in the system in a defined interval namely *t*. Stated differently, downtime is calculated periodically and in defined intervals each last for *t* units of time. In formula (1), the downtime of node *j* is divided by *t* to normalize the downtime parameter. Also w_i $(1<=i<=3)$ is the weight given to different cost function parameters and $\sum wi=1$ is always satisfied. These weights are application-dependent. We aim to select destination nodes with the minimum *DestinationCosts*, so the lower the three parameters' values are, the lower is this cost.

The load of each node *j* is defined as the summation of its data items loads, i.e., loadj = $\sum_{k=1}^{m} load_k$, in which *m* is the number of *j*'s items. The stored items in each node are either the items whose keys fall in the range of IDs for which the node is responsible, or the items that are replicated in the node. The load of each object *k* is defined as follows:

$$Load_k = size * r \quad (3)$$

In formula (3), we want to calculate the average amount of bytes that is transferred in each unit of time in relation to object *k*. Supposing that there are *r* requests for the object *k* in the related time unit, we average the sent bytes for these *r* requests and set the parameter size to the achieved result. However, there are some cases in which the node cannot respond to all of the received requests, so the node load may get bigger than its capacity. The parameter *r* is set to zero for any data item to which there is no access in the related unit of time. This way, popularities of items are applied via their access frequencies.

We define the node capacity as the maximum number of bytes that a node can transfer per time unit. These definitions focus on node bandwidth as it has been proved that even in remote storage applications, network bandwidth is likely to be the primary bottleneck [19]. As storage becomes cheaper and cheaper relative to bandwidth, this case is likely to become more common. A node is overloaded if its load is more than an upper threshold that is defined relevant to node capacity.

3.3. Valid Boundary Definition

Load balancing algorithms usually define one, two or sometimes even three thresholds [5,7]. In the same way, we use two thresholds in our algorithm, namely an upper threshold and a normal threshold. The value of normal threshold is less than the upper threshold for any node and they are both defined by considering each node capacity. It is shown by simulations that setting normal and upper thresholds of any node to 75% and 95% of its capacity respectively are appropriate choices. But different from other algorithms, we also introduce a new notion called valid boundary. We suffice to describe valid boundary here and defer how the mentioned thresholds and valid boundary are used to Section 4.

Valid boundary is separately defined for all nodes that are in the same directory. In other words, nodes in the same group use the same meaning for valid boundary.

As stated before, each directory contains some information about some of the system nodes. Using this information, we can have an overall estimation of system nodes' states in terms of their load. We mean that if the loads of at least half of the nodes in a directory are less than half of their capacities, then we guess that nodes have lots of unused capacities. In such case, it is better to balance the load in a way that more nodes have their loads under their normal thresholds. In other words, instead of just rescuing nodes from being overloaded, we try to reach their loads below their normal thresholds. To this end, we set the valid boundary to the normal threshold for this directory. If the above condition is not held, i.e., loads of at least half of the nodes in a directory are not less than half of their capacities, we just try to help the nodes not to be overloaded. In this case, valid boundary is set to the upper threshold for this directory's nodes.

By setting valid boundary to normal threshold, we stop the nodes that have been recently used in a load balancing process from becoming overloaded soon while there are unused capacities in the system. On the other hand, if lots of the nodes in a directory have used much of their capacities, we set valid boundary to the upper threshold for this directory's nodes. The reason is that it is usually hard to find some nodes to transfer the load to them in a way that load balancing process leads to a state in which all of the involved nodes have their loads under their normal thresholds.

4. LOAD BALANCING SCHEME

The area between upper load threshold and the capacity of a node works as a buffer zone before the node starts rejecting its received requests [20]. In our algorithm, a node starts running the load balancing algorithm when it notices its load is higher than its upper threshold.

Every node checks its load periodically. If it is overloaded, it puts its popular items in a list called *popular-item-list*. This list is stored separately in each node and in relation to its own items. In our algorithm, an item is popular if more than a quarter of the node load is due to that item load.

Our algorithm uses two mechanisms, namely nodes moving and replication to balance the load between nodes using the popularities of items. As replication is always followed by its own extra overheads, we try to postpone replication as much as possible while keeping the nodes in acceptable load states by applying the node movement mechanism. But in case of necessity, we make use of replication too. Node movement, replication, and search mechanisms are explained separately in detail below.

4.1. Node Movement

Node movement is done when one of the following cases arises:

1. A node gets overloaded due to the high popularity of more than one of its items.
2. A node gets overloaded because of high amount of data items put on it while none of them is highly popular. In this case the popular-item-list for this is empty.

3. A node gets overloaded while there is only one item in its popular-item-list. This item key is not equal to node's key and also the node capacity is less than half of the average nodes' capacities in the related directory.

To postpone replication as much as possible, when there is more than one popular item in an overloaded node, we try to balance its load by moving some of its popular items to another node. Also when a node gets overloaded due to excess number of assigned items to it while none of them is highly popular, we apply node movement to move some of these items to other nodes.

For the case that there is only one popular item in an overloaded node's *popular-item-list*, it is probable that due to its increasing popularity rate, moving this item to another node causes that node to get overloaded too. But considering system heterogeneity, it is possible that this node's overloading be much more due to its low capacity and not because of the great number of requests for the so-called popular item. So to delay replication, we balance the load of nodes even in this case by node movement if it is possible. To this end, we use nodes' capacities information that is stored in each directory to estimate the average capacity of nodes in the system *(C_{avg})*. When a node is overloaded while there is only one item in its *popular-item-list*, we compare its capacity with C_{avg}. If the overloaded node capacity is less than half of C_{avg}, we guess that its overloading is probably due to its limited capacity and the only item in the popular-item-list is not really a popular one. If this item key is the same as the node key, we cannot move it to another node; in this case, the only remained solution is replicating this item. But if the item key is different from the node key, we balance the load by the node movement mechanism.

If a node *n* is overloaded due to one of the above three conditions, it sends a request to its relevant directory asking it to find proper nodes to move its load. Each heavy node can balance its load with more than one node, i.e., it is possible that the load balancing process is carried out more than once and in each round a different node is selected by directory to balance the load. The selected nodes should leave their previous locations in the overlay and join at the new locations specified by the n's directory. We call these points the "split points".

Selection of each proper node is done in four steps. In the first step, the overloaded node's directory searches in its stored information, calculates the destination cost function stated in Section 3 for each of its entries, and selects a node with the minimum cost. The selected node *m* should move to a specified split point, so all of its assigned keys should be reassigned to its successor.

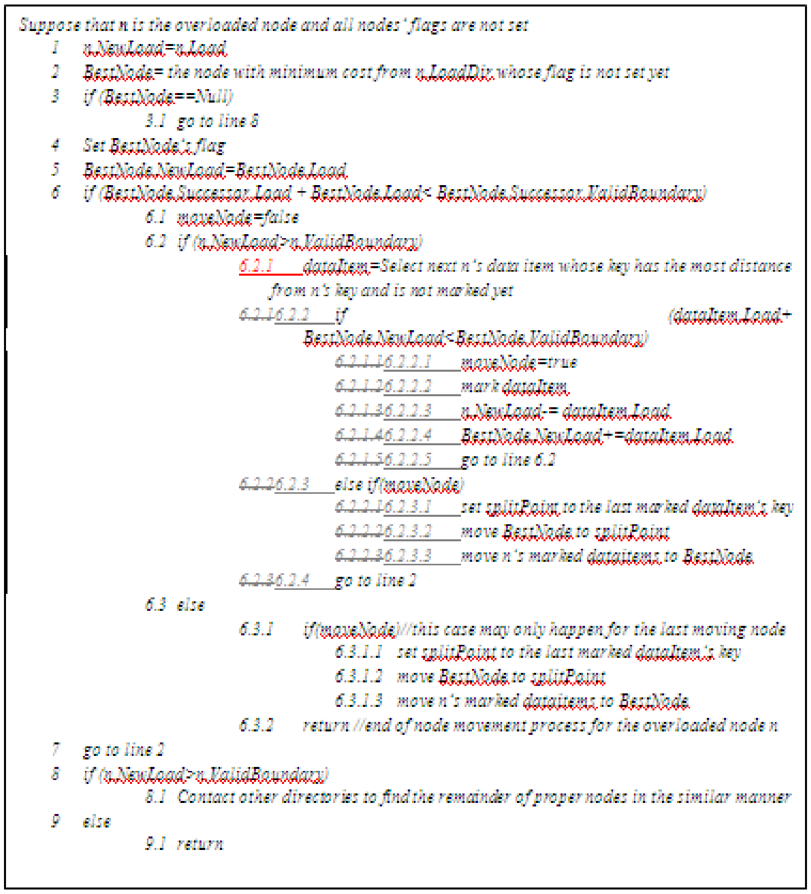
In the second step, the directory checks that the reassignment process does not end up with *m*'s successor load to exceed from its valid boundary value. If so, *m* is selected as the first proper node to leave and rejoin the system at the specified split point. If the above condition is not satisfied, the selection process is repeated from the first step to find the next minimum cost node.

In the third step, the directory checks how much of *n*'s load can be moved to *m* such that *m*'s new load does not exceed its valid boundary. To do this, the overloaded node's directory selects some of *n*'s data item, starting from the item whose key has the most distance from overloaded node key and checks the loads of *m* and *n* after moving each of these items. If the load of node *n* is still bigger than its valid boundary and also the load of node *m* is less than its valid boundary, the process of item selection continues in the same flow. In fact, no item is really moved in this step, but only the imposed load of each selected item is decreased from the current load of *n* and is added to the current load of *m*; then if the result of this summation is less than *m*'s valid boundary, this selected item is marked to be moved in the next step. We call the result of this summation *m*'s new load. This step terminates when each of the above conditions is not satisfied,

i.e. the selection of data items leads to *m*’s new load exceeds its valid boundary or the load of *n* gets less that its valid boundary. In this case the last data item is not marked and the split point is set to the last marked data item key.
The real movement of data items is done in the fourth step and after rejoining of *m* at the split point that was defined in the third step.

The process of moving the load of *n* to other nodes continues in the same flow if *n*’s load is still more than its valid boundary by selecting the next minimum cost node. The end of this flow is when *n* load reaches to its valid boundary or the only remained data item in *n* is the one whose key is equal to overloaded node key. In the latter case, the only remained data item is put in the node popular-item-list in the next execution of algorithm and this node gets an eligible candidate for the second load balancing mechanism.

If necessary, this directory can connect to other directories and selects a proper node from them. To prevent the excess number of movements of nodes when balancing the load of each node, we can define a minimum amount of load that should be moved in any node movement operation. This way, nodes that are selected to be moved must have the ability to support at least this basic minimum amount of load, although we ignore this case for simplicity. The following algorithm shows the node movement in pseudo-code.



```
Suppose that n is the overloaded node and all nodes' flags are not set
    1   n.NewLoad=n.Load
    2   BestNode= the node with minimum cost from n.LoadDir whose flag is not set yet
    3   if (BestNode==Null)
            3.1  go to line 8
    4   Set BestNode's flag
    5   BestNode.NewLoad=BestNode.Load
    6   if (BestNode.Successor.Load + BestNode.Load< BestNode.Successor.ValidBoundary)
            6.1  moveNode=false
            6.2  if (n.NewLoad>n.ValidBoundary)
                    6.2.1    dataItem=Select next n's data item whose key has the most distance
                               from n's key and is not marked yet
                    ~~6.2.1~~6.2.2    if                                   (dataItem.Load+
                               BestNode.NewLoad<BestNode.ValidBoundary)
                            ~~6.2.1.1~~6.2.2.1    moveNode=true
                            ~~6.2.1.2~~6.2.2.2    mark dataItem
                            ~~6.2.1.3~~6.2.2.3    n.NewLoad-= dataItem.Load
                            ~~6.2.1.4~~6.2.2.4    BestNode.NewLoad+=dataItem.Load
                            ~~6.2.1.5~~6.2.2.5    go to line 6.2
                    ~~6.2.2~~6.2.3    else if(moveNode)
                            ~~6.2.2.1~~6.2.3.1    set splitPoint to the last marked dataItem's key
                            ~~6.2.2.2~~6.2.3.2    move BestNode to splitPoint
                            ~~6.2.2.3~~6.2.3.3    move n's marked dataitems to BestNode
                    ~~6.2.3~~6.2.4    go to line 2
            6.3  else
                    6.3.1    if(moveNode)//this case may only happen for the last moving node
                            6.3.1.1  set splitPoint to the last marked dataItem's key
                            6.3.1.2  move BestNode to splitPoint
                            6.3.1.3  move n's marked dataitems to BestNode
                    6.3.2    return //end of node movement process for the overloaded node n
    7   go to line 2
    8   if (n.NewLoad>n.ValidBoundary)
            8.1  Contact other directories to find the remainder of proper nodes in the similar manner
    9   else
            9.1  return
```

The only remaining question is what happens if we try to find one node with a lot of unused capacity and move all the extra load of an overloaded node to it, i.e., using single node load balancing instead of multiple nodes load balancing. This way, node movement process is done only once for each overloaded node and it seems less overhead is imposed on the system. But as we show later in the simulation section, usually finding only one node requires much searching for that node which causes the load balancing process to become very slow.

4.2. Replication

If a node is overloaded because of the high popularity of one of its items while its capacity is not less than half of the average capacity of nodes in the system (C_{avg}), we avoid repetition of overloading due to this item by replicating it. In other words, supposing that there are a lot of requests for the only item in popular-item-list of a node while the node capacity is not so low compared to other system nodes, it is better to replicate this item so that its related load is distributed between the replicas.

Consider that node *n* is overloaded due to the above condition. If $load_n/C_{avg} = k$, we use this ratio to specify the number of replicas for the related data item. In other words, if *n*'s load is *k* times bigger than C_{avg} and *k* is an integer, we should redistribute the load of *n* on *k* nodes by replicating the related popular data item on *k-1* nodes other than *n* itself. In this case, each of the replicas should have the ability to support an extra load of at least $load_n/k$. If *k* is not an integer and assuming that *k*' is the lower bound (floor) of *k*, we replicate the related popular data item on *k*' nodes other than *n* itself and each of the replicas should have the ability to support an extra load of at least $load_n /(k'+1)$. If $1/2<=k<=1$ is held, we replicate the related data item only on one node other than *n* itself.

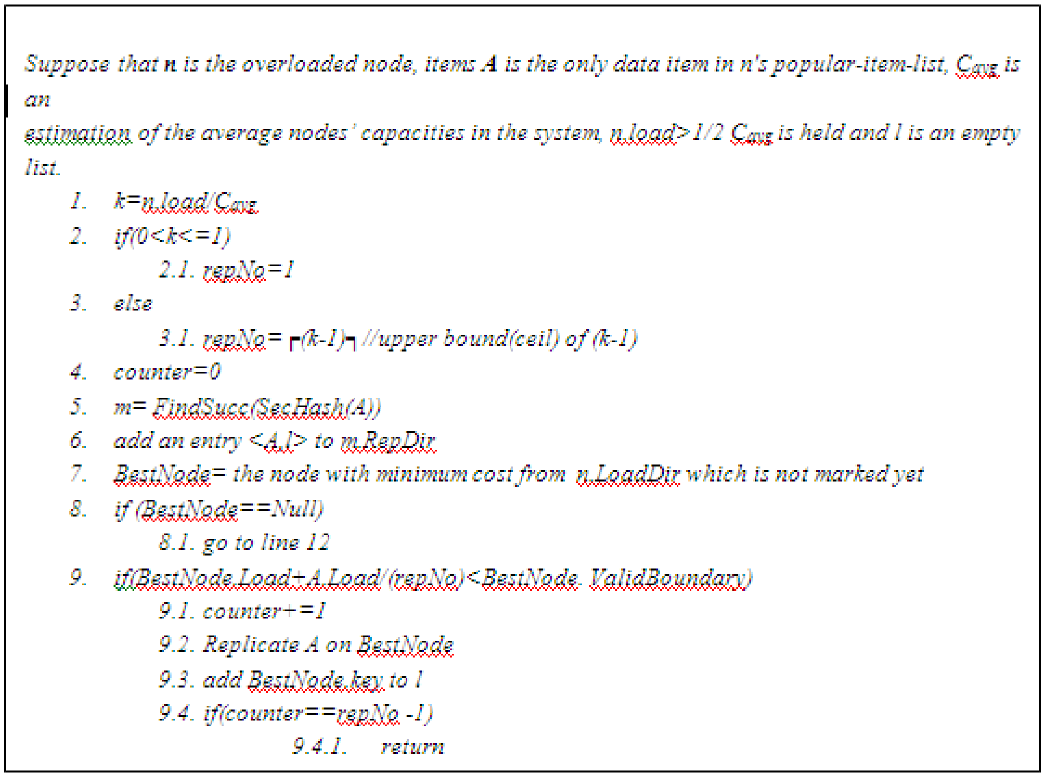
For the purpose of replication, we use a second hash function called *SecHash* and also a set of replication directories each called *RepDir*. Each entry of these directories consists of two parts, namely the name of replicated data item and a list of replication destinations. Creation of replication directories is done dynamically throughout system operation. If a node wants to replicate one of its items named *A*, it should do an operation similar to searching to find the *RepDir* where it should add an entry, but instead of searching for successor of *FirstHash(A),* it searches for successor of *SecHash(A).* Any node that stores a *RepDir* should distribute the received queries for the replicated data items among their replicas.

The replication destinations are specified by the overloaded node's *LoadDir*. This directory calculates the destination cost function stated in Section 3 for each of its entries and at last selects *k-1* or *k* of the nodes that impose the minimum cost and also by moving $load_n /k$ or *,m* to them they do not change to overloaded nodes. Again this directory can connect to other directories if necessary. *A* is replicated in the found nodes in association with another field where *FirstHash(A)* is stored. This field is used during search process as we explain below.

The overloaded node can then refuse the replicated item's received requests until its load reaches its valid boundary. This way, if a node is searching for a replicated item, it may receive no result after a period of time, which means a timeout has occurred. If this is the case, the requester node uses the second hash function to find the relevant *RepDir* and reads the replication destinations from it. It is possible however that lack of response to a request during the timeout interval be due to other reasons, e.g., links failure or high network traffic. In this case, the requester node finds no relevant entry when it refers to *RepDir* and should send another request to the item owner node.

We can avoid bandwidth wasting which is due to the short timeout intervals by setting this interval dynamically and with regard to recent response times of requests. The overloaded node sends a message to the relevant *RepDir* and replica destination nodes after its load reaches its

valid boundary, so that they can remove the associated replication information. The following algorithm shows the node replication in pseudo-code.
To make the system fault tolerant, we can backup replication directories in the *b* next successors of the nodes where they are stored. The value of *b* is defined considering the fault tolerance level needed in system.



```
Suppose that n is the overloaded node, items A is the only data item in n's popular-item-list, C_avg is
an
estimation of the average nodes' capacities in the system, n.load>1/2 C_avg is held and l is an empty
list.
    1.  k=n.load/C_avg
    2.  if(0<k<=1)
            2.1. repNo=1
    3.  else
            3.1. repNo= ┌(k-1)┐ //upper bound(ceil) of (k-1)
    4.  counter=0
    5.  m= FindSucc(SecHash(A))
    6.  add an entry <A,l> to m.RepDir
    7.  BestNode= the node with minimum cost from  n.LoadDir which is not marked yet
    8.  if (BestNode==Null)
            8.1. go to line 12
    9.  if(BestNode.Load+A.Load/(repNo)<BestNode. ValidBoundary)
            9.1. counter+=1
            9.2. Replicate A on BestNode
            9.3. add BestNode.key to l
            9.4. if(counter==repNo -1)
                    9.4.1.    return
```

4.3. Search Mechanism

Every time a node receives a request with key *k*, it checks whether *k* falls in the interval it is responsible for or not. If it is the case, this node returns the requested item. Otherwise, the node should forward the request to another node with respect to its finger table [2], but in our algorithm this step is delayed by another step in which the node checks whether a replica of the searched item is stored in itself. Since intermediate nodes have no information about the name of requested items during search process and work only with hashed keys, the node checks the requested item key with the fields stored on it in association with the replicated items, namely their *FirstHash* values. If no match is found, it forwards the request to another node with respect to its finger table, otherwise it responds to the requester node itself.

5. SELECTING CAPABLE NODES AS DIRECTORY STORES

In our algorithm we use more capable nodes to store directories in them and also using them as the director in load balancing and conflict resolution processes. In this section we first define capability metrics and then present a formula to rank nodes and select more capable nodes.

5.1. Selection Metrics

The first metric is the node bandwidth. In the context of data networks, bandwidth quantifies the data rate at which a network link or a network path can transfer [21]. The owned bandwidth of each node is not constant implying that although communication links of a node have constant hardware capacities, their free capacities highly depends on node load. This leads us to define another parameter namely $Bandwidth_{avail}$ which is calculated via (4).

$$Bandwidtha_{vail}=capacity - load \qquad (4)$$

Uptime is another metric. A node's uptime is defined as the average of continuous time the node stays in the system. Our load balancing mechanism depends on the stored information in directories. But since nodes in P2P systems can leave the system at any time, it is important to store directories in the nodes that stay in the system longer. Nodes with high values of uptime are more likely to stay longer in the system [22].

Each directory contains different kinds of information about some system nodes for the purpose of load balancing and replication. So there should be sufficient remaining memory in the nodes that are selected as directory stores. We thus consider the free memories of nodes as the third metric.

We use these defined metrics to specify where the directories should be stored. To normalize our metrics, we apply the following formula to calculate a node *i* score and then rank nodes based on their scores.

$$Score_i=w_1 * bandwidth_{avail\ i}/bandwidth_{max}+w_2 * uptime_i /uptime_{max}+w_3 * memory_i/memory_{max} \quad (5)$$

In formula (5), we have divided each node's bandwidth, uptime and memory by a related maximum value. This value is in fact the maximum achieved value among the related gathered information. Also w_i $(1<=i<=3)$ is the weight given to different parameters of score function and $\sum wi=1$ should be always satisfied. These weights are application-defined; for example, if nodes leave and join the system frequently, we should give a higher value to w_2. We explain the selection process in more details next.

5.2. Selection Process

The P2P system we have assumed consists of two overlay networks namely an overlay where all system nodes are stored and another overlay in which only directory store nodes exist. We call these two overlays *first level* and *second level* overlays, respectively. Nodes identifiers in the second level are the same as their first level identifiers.

Taking into account that comparing score values among all system nodes is not actually possible in large scale P2P systems, directory store nodes are selected as follows.

For each pointer node *p*, the pointer node after *p* in the identified space is called *p*'s *Next*. To select directory store nodes, pointer nodes periodically execute a procedure *called Second-Level-Node-Selection*. Each pointer node calculates the score function for a limited number of randomly chosen nodes called candidate nodes whose identifiers fall between its own identifier and its *Next* node identifier. Score is also calculated for the node which is already the directory store of the mentioned pointer. Finally, the node whose score has the maximum value among other comparable nodes is selected as the related directory store for the mentioned pointer node. Because score value for any node is different in variant system conditions and also candidate nodes are selected randomly, it is possible that a new node is selected in each round of *Second-Level-Node-Selection* execution. This is more probable in the early operations of the system. But as time passes, this changing rate gets lower and lower. Periodic execution of *Second-Level-Node-Selection* can help to give the chance to newly entered nodes to store directories.

First level nodes should identify second level nodes. One way is to add another set of routing tables to each node whose entries only consist of second level nodes. This solution imposes high overhead on all system nodes. A more reasonable solution is to store a field called *DirDest* in each pointer node where the identifier of a related directory store node is saved. If a pointer node decides to leave the system, it should first give its stored *DirDest* field to its successor.

As we formerly explained, each node can find its related pointer node through the *PionterNo* that is passed to it upon joining the system. So this node can get aware of its relevant *LoadDir* by reading the stored identifier in *DirDest* and it can also store it somewhere in its own memory with the name of *MyDirDest* for later uses.

It is possible however that any node including second level nodes leave the system. So the stored *MyDirDest* value in a node may be invalid after a while. In this case, when this node uses its *MyDirDest* to access its directory, two cases may happen. The first is that it finds no node with the same identifier and finds out that the previous directory store node has left the system. The second case happens when the node finds the node with the same identifier as *MyDirDest* but this node no longer contains a directory. The latter case happens when a more suitable node is found for the related interval, so that the previous directory store node is replaced with the new found node. In both cases the node can again access its pointer node and find the new value from *DirDest*. To make the system fault tolerant, each pointer node can replicate its *DirDest* value to *b* of its successors. The value of *b* is defined by the fault tolerance level needed in system.

If a second level node wants to leave the system, it first should inform its related pointer node, so that this pointer node can select another proper node to store its relevant directory by executing a procedure similar to *Second-Level-Node-Selection*.

As stated previously, the number of directories is limited. This number can be set to proportionally change with the number of nodes in the system. Small number of directories makes the nodes that store directories a bottleneck. This is because many nodes want to send their information to these nodes. On the other hand, a lot of directories in the system increase system complexity and overhead. So finding the proper number of directories can improve the performance of our load balancing algorithm. We show by simulation in the next sections that even with 16 nodes in a single group, most heavy loaded nodes are successful in shedding their loads by probing only one directory.

It is possible that some node leave the system after a while, so its related stored information in the directory is no longer valid. As we explained before, nodes periodically send their information to their related directory and a second level node gets aware of an invalid directory entry if no such information is sent for this entry in the specified periods. In such cases, this entry can be deleted by the related directory store node.

To prevent excess number of second level nodes, we define a lower threshold for the minimum number of entries a second level node can have. If a second level node notices its directory entries are lower than the mentioned threshold, it informs the central authority to set the *PointerNo* of some of the nodes that want to join the system, to its related *PointerNo*.

6. Simulation Results

To evaluate our algorithm, we have developed a Java program that simulated our algorithm on a Chord structured P2P system. Our algorithm can be applied to other structured systems too, but we choose Chord as a basic and simple structured p2p system. Simulation results describe why we have preferred multiple nodes load balancing to single node load balancing in our algorithm, shows the importance of proximity factor in bandwidth consumption, and presents the superiority of our algorithm to another three well-known related algorithms cited in related work section 2.

Our simulated nodes were completely heterogeneous with different capabilities, so Pareto node capacity distribution with parameters shape=2 and scale=100 were used. In our simulated environment 1024 nodes are included which can leave or join the system at any time.

For calculating a node down time, we used its successor and predecessor nodes to ping it every 30 seconds, i.e. the variable *t* that was defined in Section 3 was set to 30 seconds. We set the weights in Formula (1) as $w_1=0.5$, $w_2=0.25$ and $w_3=0.25$. We gave more weights to the load parameter in this formula as the first condition in any load balancing process is to find some other node which is not itself overloaded. Also the defined weights in Formula (5) were set as $w_1=0.3$, $w_2=0.5$ and $w_3=0.2$. Among these three weights, we gave the maximum value to w_2 as it is very important to place directories in nodes that stay longer in the system compared to other shorter life nodes. Because memory is cheaper than bandwidth we gave more weight to bandwidth compared to memory.

We first show why we have decided to balance the load of any overloaded node with more than one node if necessary. In other words, why we have preferred multiple nodes load balancing to single node load balancing. Considering our proposed solution to estimate the average of nodes' capacities in the system (C_{avg}), we compare the load of overloaded nodes $(load_{overloaded})$ with C_{avg}. The results of simulations in Figure 2 show that increases in the ratio of $load_{overloaded}/C_{avg}$, increases the number of searches required to find a proper node to move all the extra load of this overloaded node to it. This indicates that it takes a long time just to find a proper node to balance the load of each overloaded node. To depreciate this overhead, it is reasonable for our algorithm to use more than one node (if necessary) to transfer the extra load of each overloaded node to them.

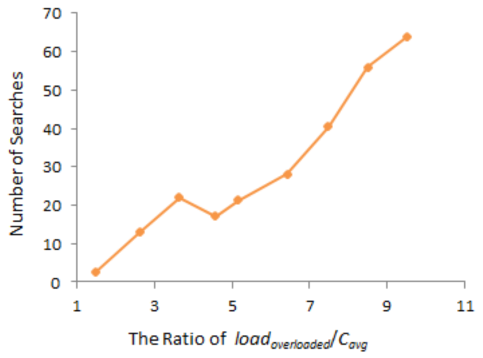


Figure 2. The rate of increase of the number of searches as $load_{overloaded}/C_{avg}$ increases

Simulations results shown in Figure 3 compares the average download times for two approaches, when multiple nodes load balancing and when single node load balancing were used. The average download time has been defined as the average (for all peer nodes) of the time elapsed since a peer requested to download a data item until the time it finished its download. Figure 3 shows the improvement (in minutes) on the average download times simulated for different query inter-arrival times (x-axis). The best results related to cases when the queries were very frequent. For example, for a query inter-arrival time of 0.2 minutes, the average download time of the data items in the multiple nodes load balancing approach was 5 minutes shorter than when the same simulation ran for the single node load balancing approach. When queries were less frequent, the improvement was negative; implying that multiple nodes load balancing is slower and less efficient than single node load balancing in case of less frequent queries.

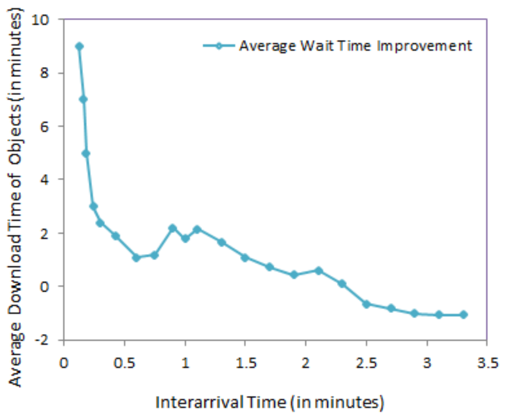


Figure 3. Download times in case of single node and multiple nodes load balancing

To demonstrate the importance of proximity in our algorithm, Figure 4 shows the bandwidth consumption of load transfer process in terms of the number of required physical hops during load balancing process. As it is shown, when we have relaxed the proximity factor, the number of required physical hops has increased.

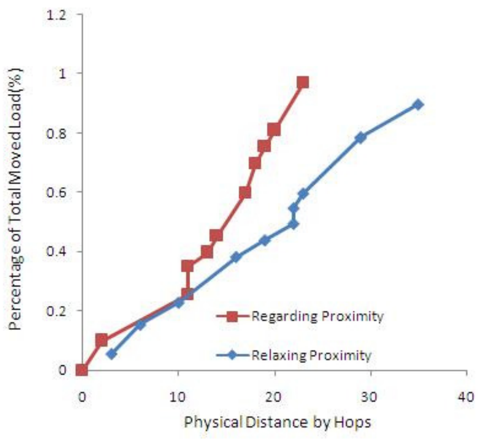


Figure 4. Changes in links delays while load balancing

Figure 5 shows the effectiveness of our algorithm by comparing it with three other related algorithms, namely Rao et al. algorithm [5], CFS algorithm [10] and log(N) virtual Server algorithm [1]. As we have mentioned previously in Section 2, Rao et al. have proposed three different schemes to balance the load using virtual servers. In our simulations, we have only considered their "one-to-one" scheme in which one node contacts a single other node per unit of time, given that their other two schemes also utilize nodes similarly.

The focus was put on the percentage of successfully routed queries for trace-driven simulations with varying loads. To this end, we have used Zipf query distribution that has first been presented by Gummadi et al. in their trace analysis of Kazaa [23] and this distribution is common to many other usages (e.g., web page file access [24] and file popularity [25]). So queries were uniformly initiated from nodes at random with destinations chosen from Zipf distribution.

Our simulations examined how the load balancing algorithms responded to different degrees of applied workload. In almost all cases, we found our algorithm performs the same or better than the other algorithms. We varied the applied query load by orders-of-magnitude and recorded the percentage of queries that reached their destination.

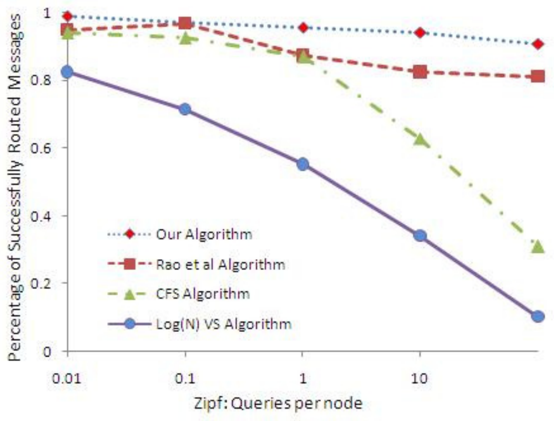


Figure 4. Percentage of successfully routed queries in trace-driven simulation with varying loads

7. CONCLUSION AND FUTURE WORKS

This paper presented a new load balancing algorithm for dynamic structured p2p systems assuming non-uniform distribution of data items in the system, heterogeneous nodes, system dynamicity, and variable popularities of objects. Also two important factors namely the downtime and proximity were considered during load transfer process. The main goal of the proposed load balancing algorithm was to increase the percentage of successfully routed queries. For the purpose of load balancing, we have designed some directories to handle the load balancing process and proposed a new approach to place these directories in more capable nodes in the system. Also we used two different mechanisms, namely node movement and replication to balance the load. Simulation results showed the superiority of our load balancing algorithm in variant load conditions in terms of the percentage of successfully routed queries.

A number of potential improvements to our algorithm deserve further studying. First, in this paper we have assumed only one bottleneck namely bandwidth. However, a system may be constrained by other resources like CPU processing capability. So our load balancing algorithm would have to be improved to handle this generalization. Secondly, our algorithm simply assumes there is no conflict when replication is done; although we postpone replication as much as possible in our algorithm, but some conflict resolution mechanism is still needed to be applied in the case of changeable data items.

REFERENCES

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications, New York, pp. 149-160, 2001.

[2] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Scott , "A Scalable Content-Addressable Network", in Applications, Technologies, Architectures, and Protocols for Computer Communications, USA, pp. 161-172, 2001.

[3] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," in Middleware, pp. 329-350, 2001.

[4] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," Technical Report UCB/CSD-01-1141, 2001.

[5] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P Systems," in Second Int'l Workshop Peer-to-Peer Systems (IPTPS '02), USA, pp. 68-79, 2003.

[6] J. Byers, J. Considine, and M. Mitzenmacher, "Simple Load Balancing for Distributed Hash Tables," in Second International Workshop, IPTPS, Berkeley, USA, pp. 80-87, 2003.

[7] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Dynamic Structured P2P Systems", in INFOCOM 2004, Hong Kong, March 2004.

[8] Narjes Soltani, Ehsan Mousavi Khaneghah, Mohsen Sharifi, Seyedeh Leili Mirtaheri, Dynamic Popularity-Aware Load Balancing Algorithm for Structured P2P Systems, International Conference on Network and Parallel Computing, September 6-8, 2012, Korea

[9] J. M. Ruhl, "Efficient Algorithms for New Computational Models", USA, Technical Report, Massachusetts Institute of Technology, 2003.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-Area Cooperative Storage with CFS", in Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01), October 2001.

[11] S. Rieche, L. Petrak, and K. Wehrle, "A Thermal-Dissipation-Based Approach for Balancing Data Load in Distributed Hash Tables", in Proc. of 29th Annual IEEE Conference on Local Computer Networks (LCN), Germany, pp. 15-23, 2004.

[12] R. Akbariani, V. Martins, E. PAcitti, and P. Valduriez, "Global Data Management (Chapter Design and Implementation of Atlas P2P Architecture)". 1st Ed., IOS Press, July 2006.

[13] Y. Xia, S. Chen, and V. Korgaonkar, "Load Balancing with Multiple Hash Functions in Peer-to-Peer Networks", in Proc. of the IEEE Int. Conf. on parallel and Distributed Systems (ICPADS), pages 411-420, Minneapolis, Minnesota, July 2006.

[14] A. Ghodsi, L. Alima, and S. Haridi, "Symmetric Replication for Structured Peer-to-Peer Systems", in Databases, Information Systems, and Peer-to-Peer Computing, pp. 74–85, 2007.

[15] M. Mitzenmacher, A. W. Richa, and R. Sitaraman, "The Power of Two Random Choices: A Survey of Techniques and Results," in Handbook of Randomized Computing, USA, pp. 255-312, 2000.

[16] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach, "Secure Routing for Structured Peer-to-Peer Overlay Networks", in ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation , New York, USA, pp. 299-314, 2002.

[17] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "Do Incentives Build Robustness in Bittorrent?", in NSDI'07 Proceedings of the 4th USENIX conference on Networked systems design & implementation , Cambridge, MA, pp. 1-14, 2007.

[18] M. Costa, M. Castro, A. Rowstron, and P. Key, "PIC: Practical Internet Coordinates for Distance Estimation", in Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04), Tokyo, Japan , 2004.

[19] C. Blake and R. Rodrigues, "High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two", in Proceedings of HotOS IX, Lihue, HI, May 2003.

[20] E. Pournaras, G. Exarchakos b, and N. Antonopoulos, "Load-Driven Neighbourhood Reconfiguration of Gnutella Overlay ", in Computer Communications, vol. 31, no. 13, pp. 3030-3039, Feb. 2008.
[21] R. Prasad, C. Dovrolis, M. Murray, and K. C. Claffy, “Bandwidth Estimation: Metrics, Measurement Techniques, and Tools”, in Network, IEEE, vol. 17, no. 6, pp. 27–35, 2003.
[22] F. E. Bustamante and Y. Qiao, "Designing Less-structured P2P Systems for the expected high churn ", in IEEE/ACM Transactions on Networking , vol. 16, no. 3, pp. 617-627, Jun. 2008.
[23] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan, “Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload”, in Proceedings of the 19th ACM SOSP, Bolton Landing, NY, October 2003.
[24] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol”, in IEEE/ACM Transactions on Networking, 2000.
[25] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, “Passive NFS tracing of Email Workloads”, in Proceedings of the 2003 USENIX Conference on File and Storage Technology, San Francisco, CA, March 2003.

Authors

Mohsen Sharifi is Professor of Software Engineering, in Computer Engineering Department of Iran University of Science and Technology. He directs a distributed system software research group and laboratory. His main interest is in the development of distributed systems, solutions, and applications, particularly for use in various fields of science. He has developed a high performance scalable cluster solution comprising any number of ordinary PCs for use in scientific applications requiring high performance and availability. The development of a true distributed operating system is on top of his wish list. He received his B.Sc., M.Sc. and Ph.D. in Computer Science from the Victoria University of Manchester in the United Kingdom in 1982, 1986, and 1990, respectively.

Narjes Soltani is M.Sc. graduate student of computer engineering in Iran University of Science and Technology (IUST). She received her bachelor’s degree in Computer Engineering from Shahid Bahonar University of Kerman, Iran, in 2009. Her research interests are in the areas of distributed and parallel systems and peer-to-peer computing.