Journal of Computer Science 10 (6): 948-960, 2014
ISSN: 1549-3636
© 2014 Science Publications
doi:10.3844/jcssp.2014.948.960 Published Online 10 (6) 2014 (http://www.thescipub.com/jcs.toc)

USING MOBILE AGENTS FOR LOAD BALANCING IN PEER-TO-PEER SYSTEMS HOSTING VIRTUAL SERVERS

VijayaKumar G. Dhas, S. Saibharath and V. Rhymend Uthariaraj

Anna University, Chennai, India

Received 2013-12-16; Revised 2013-12-23; Accepted 2014-01-27

ABSTRACT

This study proposes a novel load-balancing algorithm for managing virtual servers in a Peer-To-Peer (P2P) system through mobile agents. The proposed algorithm is implemented in a fully decentralized manner for a structured P2P system. It uses mobile agents and is independent of the geometry of the P2P auxiliary networks. The load-balancing algorithm effectively reduces the load imbalance of the system using the load per unit capacity derived by the mobile agents. A unique feature of the proposed algorithm is the mutual swapping of virtual servers between overloaded and underloaded peers to efficiently use the available resources. The proposed solution has been verified in a P2P environment consisting of peers and embedded glassfish server instances, created dynamically to act as virtual servers.

Keywords: Load Balance, Virtual Servers, Heterogeneity, P2P Systems

1. INTRODUCTION

Load balancing is an important factor to consider when optimizing productivity in Peer-to-Peer (P2P) computing. The load must be distributed among peers based on their ability to get better throughput (Wang and Vanninen, 2006; Zou *et al*., 2002). Instead of transferring tasks between peers, the trend for some time has been to migrate virtual servers from one peer to another. In this case, each peer hosts a number of virtual servers. To balance the load of the system, the virtual servers are moved from an overloaded to an underloaded peer (Rao *et al*., 2003; Li and Shao, 2011; Wang and Vanninen, 2006; Zou *et al*., 2002; Hsiao *et al*., 2011; Godfrey *et al*., 2004). In earlier work, the virtual servers were transferred from overloaded to underloaded peers in only one direction (Hsiao *et al*., 2011). The work carried out by (Wang *et al*., 2004; Zhu and Hu, 2005) considered a node k that has a specified target load T_k, where T_k is less than C_k (the capacity of k). The node k can accept virtual servers only up to the target load T_k. Ideally, T_k should be the product of A and C_k (Wang *et al*., 2004), where A is the load per unit capacity of the system. In this case, k manages its load proportional to its capacity. Although this load balances the system, the load imbalance factor is not effectively minimized. Sometimes an overloaded peer cannot migrate a virtual server to an underloaded peer, because if the underloaded peer accepted the load it would become overloaded. The primary aim is to reduce the load of the overloaded peer until it becomes underloaded. Once an overloaded peer becomes underloaded, it remains that way until the next load balancing cycle, which is an inefficient use of resources. The unique feature of the proposed algorithm is the mutual swapping of virtual servers between the overloaded and underloaded peer, which overcomes the above limitation.

Our primary aim is to reduce the load imbalance factor of the whole system. We propose a novel load balancing algorithm that exchanges virtual servers between overloaded and underloaded peers. This primarily helps to reduce the load imbalance factor of both overloaded and underloaded peers. Mobile Agents (MAs) are used to find the load per unit capacity and help overloaded peers to find underloaded peers to which to migrate their virtual servers. Using this method, the load imbalance of the whole system is significantly reduced. A virtual server can be migrated to the underloaded peer and some smaller virtual servers hosted

Corresponding Author: VijayaKumar G. Dhas, Anna University, Chennai, India

by the underloaded peer can be sent to the overloaded peer to create balance. This minimizes the load imbalanced factor. Previous studies have assumed only one bottleneck resource in the system (Wang *et al*., 2004; Zhu and Hu, 2005). The study carried out in (Hsiao *et al*., 2011) also assumed only one bottleneck resource in the system, but it failed to assume a particular resource constraint. We have considered a specific resource as a constraint, namely the number of active TCP ports. As in earlier studies, we attempt to minimize the movement cost. In his pioneering work (Lamport, 1978), Lamport proposed that events must be ordered in a distributed multiprocessor system. The ordering of events using logical clocks removes most of the unexpected anomalous behaviors (Lamport, 1978). We apply this concept when handling requests to move virtual servers from overloaded to underloaded peers.

The algorithm proposed addresses the challenge of load balancing. It reduces the load imbalance factor and movement cost. We also consider Lamport's ordering of events when moving virtual servers. We have implemented these concepts with a 1-N scheme and tested the algorithm in a P2P environment where dynamic embedded glassfish servers instances were used as virtual servers. We experimented using the number of active TCP ports as a resource constraint in the proposed load-balancing algorithm.

The remainder of this study is organized as follows. In section 2, we review some related work. Section 3 presents the proposed framework for load balancing in structured P2P networks. In section 4, we explain the structure of our experiments. We present results from an experimental study in section 5. Section 6 discusses how the load is captured, using a TCP port as a resource constraint. Section 7 presents experimental results obtained based on load imbalance factor and movement cost. It also discusses the pros and cons of the proposed algorithm. Our conclusions are found in section 8.

2. RELATED WORK

Load balancing is done by migrating the virtual servers from an overloaded to underloaded peer and has been discussed in numerous studies (Godfrey and Stoica, 2005; Ledlie and Seltzer, 2005; Wang *et al*., 2004). There are three schemes for overloaded to underloaded peer interaction: 1-1, 1-N and N-N. The simplest is the 1-1 scheme, which involves less computation and overheads but does not select the best possible peer to host the virtual server (Rao *et al*., 2003). In the 1-N scheme the overloaded peer can select the best possible underloaded peer to host the virtual server. The N-N scheme uses a directory to share information. It reduces the load imbalance but becomes similar to a centralized directory scheme. N-N is very useful when the overloaded peers have information about other overloaded peers in the proximity. The overloaded peer can mutually exchange the details of the underloaded peers that it has interacted with in the past and present.

Wang and Vanninen (2006) and Zou *et al*. (2002) focused on maximizing the throughput of the system and reducing message overheads and considered the CPU and memory capacities as load parameters. The maximum number of socket connections possible per unit time is considered to be a parameter. Although they tried to maximize the throughput of the system, they did not consider this parameter. In addition, they did not try to balance the load over the whole system so that every peer reached its threshold level.

Hsiao *et al*. (2011) tried to reduce the load imbalance factor by transferring virtual servers from overloaded to underloaded peers. They tried to reduce the movement cost, but did not try to minimize the load imbalance factor. Their method used the virtual server concept for load balancing, by assigning load to each object entering the system (Hsiao *et al*., 2011; Godfrey *et al*., 2004). However, their method assumed that there was only one bottleneck resource in the system and left the multiple resources as future work. Minimizing the load imbalance factor requires more movement cost. Although better balancing can be achieved by higher movement cost, the load imbalance factor must not be too high compared with the system's bandwidth.

HiGLoB is a histogram-based global load balancing framework (Vu *et al*., 2009). It uses a load-balancing manager to redistribute the load among peers. However, this load-balancing manager is centralized and cannot be used in a fully decentralized P2P environment.

In "single ID per node" only one popular object can be stored in a single node. It failed because of the popularity of some objects in the destination node (Karger and Ruhl, 2004; Serbu *et al*., 2007). "Multiple ID per node" was then introduced, but it too ignored the popularity of objects (Rao *et al*., 2003). Li and Shao (2011) and Nehra *et al*. (2007) used MAs to collect, analyze and locate peers. These MAs identify each peer as overloaded or underloaded. The MAs walk through the entire network and try to load balance each system by migrating the virtual server. The MAs are not lightweight, as they must perform a lot of processing.

They are responsible for coordinating the transfer of workload between overloaded and underloaded peers.

Shen and Xu (2007) have considered proximity of the peers in load balancing. There is a mismatch in the proximity abstraction of logical and physical location of peers. The mismatch makes load balancing proposed by (Shen and Xu, 2007) suitable only for specific topology. Hsiao *et al*. (2011) have pointed out that the load balancing problem need not rely on auxillary tree networks and should be independent of the geometry of the P2P Substrate.

Local load information aggregation is done by sending messages with a time to live TTL value, which is decremented by one when it is received by the adjacent peer (Li and Xie, 2006). The load per unit capacity is calculated using this information. The limitation of this approach is that it leads to larger message overheads. As each peer collects information, it forwards the message to the next peer. The same peers can be visited multiple times and the calculated load per unit capacity is not accurate because they are counted many times. Therefore, a peer is covered multiple times by different messages. When using the local neighborhood method, only he bandwidth involved is minimized. However, the problem of reducing the load imbalance is not given importance. Even if one tries to reduce the bandwidth latency, the load imbalance factor must also be considered.

3. PROPOSED SYSTEM

We propose an innovative mechanism that uses MAs to control virtual servers. The migration of the virtual servers is used to balance the loads in structured P2P systems.

Consider a P2P system, with a set of virtual servers V and a set of peers, N, participating in the system. Let Lv be the load of the virtual server v, where v is a subset of V. The load of the peer $Load_i$ is the sum of the loads of all virtual servers hosted by the peer. Let Ci be the capacity of the peer i. The load per unit capacity (Lpc) of the system is defined as the sum of loads of all virtual servers divided by the sum of the capacities of all the peers. Some parameters used can be found in **Table 1**.

3.1. Calculating Load Per Unit Capacity

The sum of loads of all virtual servers is defined as:

$$\text{Sum of loads} = \sum_{i=1}^{\text{n peers}} \sum_{v=1}^{\text{k virtual servers}} L_{i,v}$$

The load per unit capacity is defined as:

$$\text{Lpc} = \frac{\text{Sum of loads}}{\sum_{i=1}^{\text{n peers}} C_i}$$

The load imbalance factor of a particular:

$$\left| \text{Peer}_i - \text{Ti} - \sum_{v=1}^{\text{k virtual server}} L_{i,v} \right|$$

The load imbalance factor for the whole system is defined as:

$$\text{Load imbalance} = \sum_{i=1}^{\text{n peers}} \left(\left| \text{Ti} - \sum_{v=1}^{\text{K virtual servers}} L_{i,v} \right| \right)$$

MAs are used to monitor groups of several peers. The MA obtains the total load contributed by all the virtual servers hosted by a peer and the capacity of the peer. The various MAs share this information and find the total Load per unit capacity (Lpc) of the system. The load per unit capacity is given to all the peers in the system. Based on the Lpc, each peer calculates its threshold load value T_i, which is proportional to the capacity of the peer. The threshold load value, T_i and the load of the $peer_i$ are compared. The peer is overloaded if the load of the $peer_i$ is greater than the threshold load value T_i. The status of the peer (overloaded or underloaded) is communicated to the MA. For each $peer_i$, the MA stores the sum of loads, capacity of each peer and status of the peer as overloaded or underloaded. MAs periodically update the load per unit capacity of the system.

The proposed algorithm is event based and the following events takes place.

On receipt of load per unit capacity (Lpc) at $Peer_i$
    Check status of the Peer
    If overloaded
        Send request message for load migration
On receiving request message by the under loaded peer
    Process the message and send response message
On receiving the response message by overloaded peer
    Add to message queue
    On time out at over loaded peer
        Process queue and send accept message
    On receiving accept message at under loaded peer
        Carry out exchange of virtual servers

The above algorithm is discussed in detail in this section.

Table 1. Notations frequently used

T_i	Threshold of $peer_i$	R_i	Set of virtual servers to be returned back from underloaded to overloaded peers
A	Load per unit capacity	L_v	Load of virtual server v
C_i	Capacity of $peer_i$	V	Virtual server
$LOAD_i$	Load of $peer_i$	Lpc	Load per unit capacity
MA	Mobile agent		

Module 1

On receipt of a load per unit capacity for $peer_i$ from a MA.

When the peer finds that it is overloaded, some of the virtual servers whose load Lv is minimal need to be migrated until its load becomes less than the threshold load value, T_i. Let Δ_i be the difference between the new load on $peer_i$ and T_i, after the required migration. That is, $peer_i$ is now underloaded by Δ_i. This $peer_i$ can now accept a load value up to Δ_i to become load balanced, thereby reducing the load imbalance factor.When peers update their sum of loads, the status is refreshed. So the overloaded peer finds the underloaded peer through the MAs. The overloaded peer sends L_v and Δ_i to the underloaded peers identified by the MA. After sending the requests to the underloaded peer, the overloaded peer waits for responses. The above process is illustrated by the algorithm for an overloaded peer. This algorithm is activated when a peer becomes overloaded.

```
{
 Ti = Lpc*Ci
 if loadi > Ti
        Status = overloaded
 else if loadi < Ti
        Status=underloaded
 if (peer is overloaded)
        while loadi >Ti
                Ui = min {Li, v}
                //V hosted in peeri
                Δl = Ti-(Loadi-Li,v)
                Send (Ui, Δi , peerids)
//Sends request to many underloaded peers. The peer ids of the underloaded peer is given by MA
}
```

Module 2

On receiving a request message by the under loaded $peer_j$.

The underloaded peers can reply with a request to accept virtual servers whose sum of load is less than Δ_i. This helps to load balance both the overloaded and underloaded peers after the virtual servers have been exchanged. The load imbalances of both are minimized and both loads converge to their threshold.

The underloaded peer receives the request to accept virtual servers, with a maximum return of Δ_i. The underloaded peer checks that it will not exceed the threshold value if it accepts then sends a response (C(response)) to the overloaded peer. It also attaches its load imbalance, capacity and R_j value (set as null). The R_j value is the set of virtual servers whose sum of load value is less than or equal to Δ_i. Let α be the difference between the new load on the $peer_j$ and the threshold load value T_j. If α of $peer_j$ is greater than Δ of $peer_i$, then the request is rejected. Otherwise, the formload function is called to find the combinational set of virtual servers that can be sent from the underloaded $peer_j$ to the overloaded $peer_i$. If the formload function cannot find the combinational set of virtual servers, then it will set R_j to null. If R_j is null, then the request from the overloaded $peer_i$ is rejected. If R_j is not null, then the request from the overloaded $peer_i$ is accepted. The above process is illustrated by the algorithm for the underloaded peer. This algorithm is activated when a peer is underloaded.

```
{
  if (loadj + Li,v ≤ Tj)
        Send (peerid, loadimbalance, capacity,
Rj = Null)          // sending response
  else
        {
        α = loadj + Li,v – Tj
        if α > Δi
                reject request
        else
          Rj = call formload function(α, Δi)
          if ( Rj = = Null )
                Reject request//unable to find a peer
          else
                Send (peerid,loadimbalance,capacity,
                   Rj) //sending response
        }
   }
```

Module 2a

Form the load and selecting virtual servers to be sent from the underloaded $peer_j$ to the overloaded $peer_i$ in case of swapping [Minimum α and maximum Δ_i].

The formload function selects the set of virtual servers which can be transferred on return from underloaded peer to the overloaded peer on accepting the virtual server from the overloaded peer. This helps to reduce the load imbalance factor of both peers. The array Ld contains all the loads of virtual servers that are hosted by the underloaded peer whose load value is lesser than or equal to delta and n is the number of virtual servers. Initially, R_j is set to null. K is a two dimensional array that tracks the maximum load value which can be returned as a set of virtual servers. The best possible combination of virtual servers whose sum is less than or equal to delta is selected. Variables i and w points to the virtual server and the load value for which the best combination of virtual server is selected. K[n, Δ_i] is the sum of load for a set of virtual servers that can be sent to the overloaded peer. If K[n,w] is greater than alpha, it is rejected because the underloaded peer becomes overloaded if the exchange process happens. If the load to be formed is zero or no virtual server is considered, K(i,w) is zero. The array is constructed in such a way that if the load of the virtual server is greater than the load to be formed (w), then the load of the previous virtual server (i-1) is retained (i.e., K(i,w) = K(i-1,w)). If the load of the virtual server is less than the load to be formed (w), then K(i,w) is equal to the maximum of the load formed in the previous virtual server (K(i-1,w)) and the load of the current virtual server (ld[i-1]) plus the load formed in K[i-1][w-ld[i-1]].

The above process is illustrated in the following algorithm.

Formload:

```
ld[ ] = For each Uj whose Lj,v ≤ Δi
Let n be size of array ld
declare K[n+1][ Δj +1], Rj = NULL
for I = 0 to n
 for w = 0 to Δj
  if I = = 0 || w = = 0
    K[i][w] = 0;
else if ld[i-1] ≤ w
    K[i][w] = max(ld[i-1] + K[i-1][w-
                                    ld[i-1]], K[i-1][w]);
  else
    K[i][w] = K[i-1][w];
if(K[n][w] <α)
   return NULL;
else
     size = Δi
     while n>0
   if K[n][size]! = K[n-1][size])
     Rj = Rj U Vn-1 // Vn-1 has load ld[n-1]
     size - = ld[n-1];
     decrement n
        else
         decrement n
return Rj
```

Thus, the overloaded $peer_i$ migrates its virtual server with minimal load to an underloaded $peer_j$. The overloaded peer continuously tries to migrate its minimal virtual server until it becomes load balanced, or its overall load is less than the threshold. The underloaded peer never becomes overloaded.

The set of virtual servers used to form the load is tracked as discussed below. Initially, n is the number of virtual servers on the peer. The variable "size" is set to delta. If K(i, size)! = K(i-1, size), the present virtual server has been involved in forming the load. So this virtual server is added to the set R and size is decremented by the load of the current virtual server. This process is executed in a loop until the value of n becomes zero and R contains the set of virtual servers that form the load K(n,w).

As an example, In underloaded $peer_j$, say $T_j = 50$ and Load (j) = 40. In overloaded $peer_i$, say $T_i = 50$, Load (i) = 54 and minimal virtual server load $L_{i,v} = 15$. The overloaded peer tries to migrate this virtual server. $\Delta_i = T_i-(Load_i-L_{i,v}) = 50-(54-15) = 9$. The underloaded $peer_j$ receives a request from overloaded $peer_i$ with $<L_{i,v}, \Delta_i>$ as <15, 9> respectively. The underloaded peer cannot accept V directly without return of virtual servers as $LOAD_i + L_{i,v} > T_j$.

$\alpha = LOAD_j + L_{i,v} - T_j$, $\alpha = 5$ units respectively. α is not greater than Δ_i. So formload function is called.

Module 3.

On receiving the response message by the overloaded peer

After the overloaded peer sends the minimum virtual server's load value and delta value to the underloaded peers, the overloaded peer waits for a time out to occur. While it is waiting, it saves the responses received from underloaded peers.

On receiving a response message from the overloaded $peer_i$:

```
// for each response
Response[K] = recv(peerid, loadimbalance, capacity, Rj)
Increment K.
```

Module 4.

On time out at over loaded peer.

Multiple request handling by an underloaded peer.

When requests arrive at an underloaded peer and it responds positively, only some of the overloaded peers can migrate their loads to it, or it will become too overloaded.

Two approaches to avoid this overloading are possible. First, when there is a positive response to the overloaded $peer_i$, the load of the underloaded $peer_j$ is temporarily increased to $L_{i,v}$ units. When another overloaded peer sends a request, j projects its load as $Load_j + \sum L_{i,v}$, even though it has not yet been selected to host additional virtual servers. Then the underloaded $peer_j$ cannot become overloaded. When this peer is not selected to host a virtual peer by the overloaded $peer_k$, it decreases its load by $L_{k,v}$ units. However, this is very conservative approach, because not all the positive responses will succeed and this peer will not be selected as the best peer by the overloaded peers.

The second approach orders events (Lamport, 1978). Suppose an $overloaded\_peer_1$ sends a request with L_{1v} and a positive response $C(response_1)$ is sent from the underloaded peer. Then another $overloaded\_peer_2$ sends a request (with L_{2v}), it sends a positive response $C(response_2)$. The ordering ensures that.

$$C(response_1) < C(response_2),$$

i.e., $C(response_i) < C(response_j)$ where j>i.

The $overloaded\_peer_1$ receives the response and processes it. Suppose that $overloaded\_peer_2$ replies before $overloaded\_peer_1$. The underloaded peer checks whether $C(response_2)$ is less than $C(response_1)$. If it is not, the reply is not in order. It does not host the virtual server of $overloaded\_peer_2$ until it receives the response from $overloaded\_peer_1$, unless it can host L_{1v} and L_{2v} without exceeding its threshold.

3.2. Selection of Peer for Migration

The overloaded peer collects responses from the underloaded peers, which contain load imbalance, capacity and the set of virtual servers to be exchanged with the underloaded $peer_j$. The overloaded peer first checks through the list for peers that do not need to exchange virtual servers (R_j is null). Priority is given to underloaded peers that not need to exchange any virtual servers with the overloaded peer. The underloaded peer that has the maximum product of load imbalance and capacity is selected. If there is no underloaded peer that has null R_j, it checks the remaining responses, which are from peers that need to exchange a set of virtual servers to the overloaded peer. The overloaded peer selects the underloaded peer from which is must take the minimum number of virtual servers. Module 2 ensures that the overloaded peer will not become overloaded once again, because it will not accept any servers if its load after exchanging is greater than its threshold. After the overloaded peer has selected the appropriate underloaded peer, it migrates the minimum set of virtual servers. It also sends the responses that it received from the other underloaded peers. It is possible for the underloaded peer's load to increase, because the loads of the virtual servers change when they handle other requests. In this case, responses from the other underloaded peers can be used to transfer virtual servers. The above process is illustrated by the following algorithm.

On a time out at the overloaded peer:

```
Min = -1
for each response[K] whose Rk is Null
  if (loadimbalance*capacity > min)
        Index = K
        Min = response[k].loadimbalance *
                        response[k].capacity
if min != -1 // means a response is selected
 Select the peer(response[index].peer id) for transfer
                of virtual server
 Loadi = Loadi - Lu,v
 V = V- Ui
  return
 for response[K] whose Rk is not Null
 max = delta
 sumk = ∑ response[k].Lj,v
 if sumk< max
        Index = k
        Max = sumk
Select the peer(response[index].peer id) for transfer of
virtual server
Loadi = Loadi + Rindex - Li,v
V = V- Ui
V = V U Rj
```

Module 5.

On receiving an accept message at an underloaded peer.

When the underloaded peer receives an accept message from the overloaded peer, it accepts the virtual server load. If the underloaded peer's condition has changed and it will become overloaded, the alternate best choice is used.

If there are no alternate peers available, the underloaded peer restarts the load-balancing algorithm:

```
Receive accept message
if (message = = accept message) {
 receive (peerid,Lu,v,reponse[])
 if(could not host v)
        Choose best underloaded peer from reponse[]
        and transfer
   if could not find a peer from response[]
                Restart load balancing algorithm
 else
        V = (V-Ri) U Ui
 }
```

4. ARCHITECTURE DIAGRAM

Every peer in the P2P environment hosts a number of virtual servers. Peer modules contain all the necessary algorithms. The peers share their sum of loads and the capacities. The virtual server hosted by the peer runs the application, handles and processes the requests and interacts with the peer. Each MA calculates the load per unit capacity for the set of peers it is monitoring. The load per unit capacity is shared and updated between adjacent MAs, they calculate the global load per unit capacity and inform the peer. The peer informs the MA when there is a change in load on a virtual server. The architecture diagram of the system is shown in **Fig. 1**.

4.1. MAs

A MA supervises a group of several peers. It periodically obtains the sum of load for each virtual server and the capacity of the peer.

Load per unit capacity is the sum of the load of each virtual server of every peer divided by the sum of the capacity of all peers. This Lpc is shared between MAs.Sum of loads of all virtual servers and capacities of peers which are shared among mobile agents are organized in the following manner:

```
struct load_cap {
double sum_load_vs,sum_cap;
int ma_id;
} mobile_agents[N];
```

Every MA maintains this and calculates the load per unit capacity after the information is globally shared. Instead of each agent adding all the values, it can get the sum of the loads and capacities from adjacent MAs. Therefore, this MA can add the remaining MA's collected load and capacity, find the load per unit capacity for its monitored peers and also share the sum of loads and capacities with the MA so that is communicated to other MAs. This is the only function of MAs when they are not participating in any load balancing or when the virtual servers are being moved.

4.2. MA Management

The MAs can be deleted or created. A MA failure is detected in the following way. First, the mobile agents periodically share their load per unit capacity. When none of the MAs receive a message from a particular MA, it is said to have failed. Peers that were tracked by that MA should then be monitored. Therefore, every peer maintains a status to show that it is being monitored by some MA. Secondly, if some peer's status is null, an MA has failed. Peers that are monitored by the same MA know each other's address. When an MA fails, these peers communicate among themselves and create a new MA using an established mechanism.

5. EXPERIMENTAL SETUP

Systems were connected through local area network with capacities of random access memory varying from 1 to 8 GB and processors of Intel Pentium 4 or core or core 2 duo or atom or i3 or i5. Every system runs with Ubuntu 12.04 OS and simulation was done in Java and Netbeans IDE was used. Virtual servers were dynamically set up by embedded glassfish server instances in each peer. Each instance runs the application in WAR file. In this simulation, ticket booking application runs on each instance. User request from html webpage on arriving are mapped to a random virtual server instance of a peer. In JSP, the request was mapped to the peer which hosts the virtual server through its ip address. First, the peer module handles the incoming request. It allocates the task to the virtual server which was selected in random. Suppose the virtual server was not found in the peer in which it was located, peer allocates the task to one of its virtual servers at random. The application on the virtual server handles the request, books the ticket, and updates in database. During this process, virtual server in the peer may not be found in the peer as it could have been migrated by the load balancing algorithm which was running concurrently.

Mobile agent was implemented in Java aglets which exchanges loads of virtual servers and peers with other mobile agents and calculated load per unit capacity. It also helps overloaded peers to identify underloaded peers for sending requests. One MA was allocated per two peers. Load balancing algorithm is implemented in a Java file which captures the load of each virtual server and shares the load and capacity with the mobile agent. On receiving load per unit capacity from mobile agent, the event based load balancing algorithm runs in the system.

6. EXPERIMENTAL ANALYSIS

6.1. Capturing the Load of the System

The load of the virtual server was calculated by:

- Obtaining the pid's of the virtual server
- Determining the memory and CPU utilization in the Linux environment using all the pid's
- Finding the ports used by the particular pid's

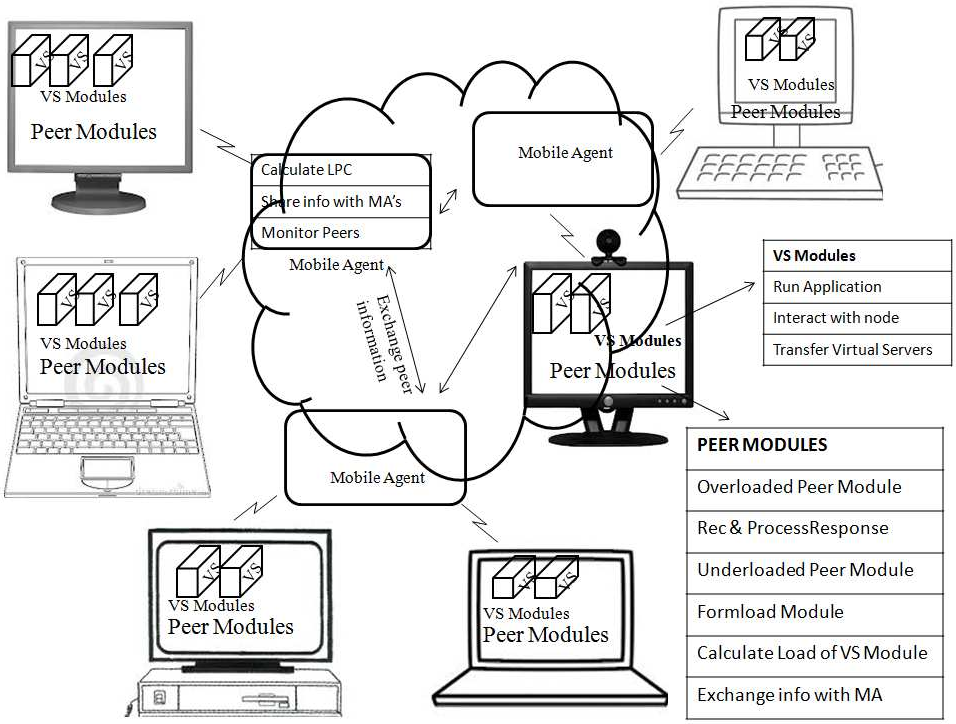

Fig. 1. Architecture diagram

Thus, we defined the load by using the above parameters. The capacity of the peer is the memory, CPU capacity, bandwidth and available open ports. We determined the load of a virtual server using its CPU consumption, memory usage and the TCP/IP connections used by the virtual server at that particular second. First, the system captured the process ids being used by the virtual server. We obtained the CPU usage of these process id's using the command %cpu, the memory usage using %mem. We determined the number of TCP/IP connections used by the virtual server and then the number ports using "/proc/pid's/". These values were periodically obtained and stored. The peer periodically updated the loads of its virtual servers. Define c1 to be the CPU consumption, c2 to be the memory usage and c3 to be the ports and TCP/IP connections. In our experiments, these were given equal weight and set to 0.33, so c1+c2+c3 = 1.

Similarly, the capacity of the system was determined using the CPU capacity, memory capacity and maximum number of TCP/IP connections. The CPU and memory capacity were static. At each period, we determined the used and free TCP/IP connections. Therefore, the load and capacity of the peer changed periodically and was monitored by a MA that calculated the new load per unit capacity and informed each peer that it monitored.

6.2. Considering Maximum TCP/IP Connections

In this simulation, we have assumed that systems have considerable CPU and memory capacities. Then the maximum possible number of socket connections, or that can be guaranteed per unit time, becomes important.

The load is dramatically increased in a system with less available TCP connections, or where there are more connections used per unit time and where there are many connections used by the virtual servers. Similarly, when fewer connections are used and more connections are available to the peer, the load decreases. The change in load helps us to better categorize the peers as overloaded or underloaded.

A system that is classified as underloaded without considering TCP/IP connections, but which has less free connections, will performance worse in terms of the number of requests handled in unit time. However, if we consider the TCP/IP connections the load of each virtual server is increased and system's overall load is increased.

Now the peer has a higher chance of becoming overloaded or load balanced. The same applies to an overloaded peer. It may have abundant free TCP/IP connections, so it can host more virtual servers and can handle more requests per second. In addition, if a peer is classified as underloaded without considering TCP/IP connections and has more available connections, it can become more underloaded with a higher load imbalance. The converse is also true. All these four cases are possible.

Loads of the peers are calculated using CPU and memory usage as load parameters and CPU, memory usage and TCP IP as load parameters simultaneously. Then the mobile agents calculate and send load per unit capacity to the peer for each of the load calculation. Of all the underloaded peers under CPU and memory usage as load parameters, 70% remain as underloaded, 18% become load balanced and 12% become overloaded when TCP IP parameter is included along with CPU and memory usage as load parameters. Similarly of all the overloaded peers when CPU and memory usage were used as load parameters, 63% remain overloaded, 24% become overloaded and 13% become load balanced when TCP IP parameter is included.

The changes to the classification of peers shown in **Fig. 2** would result in a different virtual server migration by the proposed load-balancing algorithm. More requests would be handled per unit time. We obtained the range of available open ports using the command sysctl net.ipv4.ip_local_port_range and the minimum lifetime for a TCP/IP connection using sysctl net.ipv4.tcp_fin_timeout stackoverflow.com/questions/410616/increasing-the-maximum-number-of-tcp-ip-connections-in-linux. The maximum connection that a system can guarantee per unit time is the range of open ports divided by minimum lifetime. The range in our experimental system was 32768-61000 and the minimum lifetime was 60. Therefore, it could guarantee 470 socket connections per unit time. So the maximum possible connections was considered as a load parameter, which we set to 470, 766 and 1530. The maximum possible connection was increased to 766 by increasing the open ports range to 15000-61000 and 1530 connections were made possible by reducing the minimum lifetime to 30.

We ran the proposed load-balancing algorithm, considering only the CPU, memory and the list of ports used by the system. When we did not consider the TCP connections as a parameter, the average number of possible connections for a load-balanced system was 390. It was considered load balanced because the load of the peers equaled their thresholds, although more socket connections could be used. When we did consider the TCP connections, additional virtual servers were hosted if the peer was underloaded (considering CPU, memory and available free TCP connections). The average number of connections used in a load-balanced system was then 425. We carried out the same experiment on systems with 766 and 1530 maximum possible connections. The results are shown in **Fig. 3** and demonstrate that more socket connections are used per unit of time when TCP is considered as a parameter.

6.3. Migration of Virtual Servers

Each peer's module receives requests to execute tasks. They randomly select a virtual server and assign it the task. When the task is completed, the peer module sends a reply. When a virtual server is moved from an overloaded to an underloaded peer, the application file that is running on the server and the embedded glassfish server instance creation file is transferred to the underloaded peer. The underloaded peer deploys the instance and the application is launched on the server instance. The overloaded peer's module checks all the requests or tasks that have been allocated but not successfully completed and these tasks are moved to the selected underloaded peer's module. The underloaded peer's module allocates these tasks after the virtual server has been successfully deployed.

6.4. Local Vs. Global Load Per Unit Capacity and Load Balancing by MAs

We ran two types of simulations using MAs. In the first, each MA monitored N peers. It calculates the load per unit capacity of the peers it is monitoring and informs its peers. An overloaded peer tries to migrate its virtual servers to the underloaded peers in this MA's coverage. This reduced the bandwidth latency needed to transfer the virtual server. After this initial process, the adjacent MAs share their load per unit capacity and inform the peers about the new load per unit capacity. Now, the overloaded peer first attempts to migrate the virtual server to the underloaded peer in the same MA's coverage. If this is not possible, it tries to find the nearest underloaded peer in the adjacent MA's coverage. This process repeats. By using this scheme, the required overheads and bandwidth are reduced.

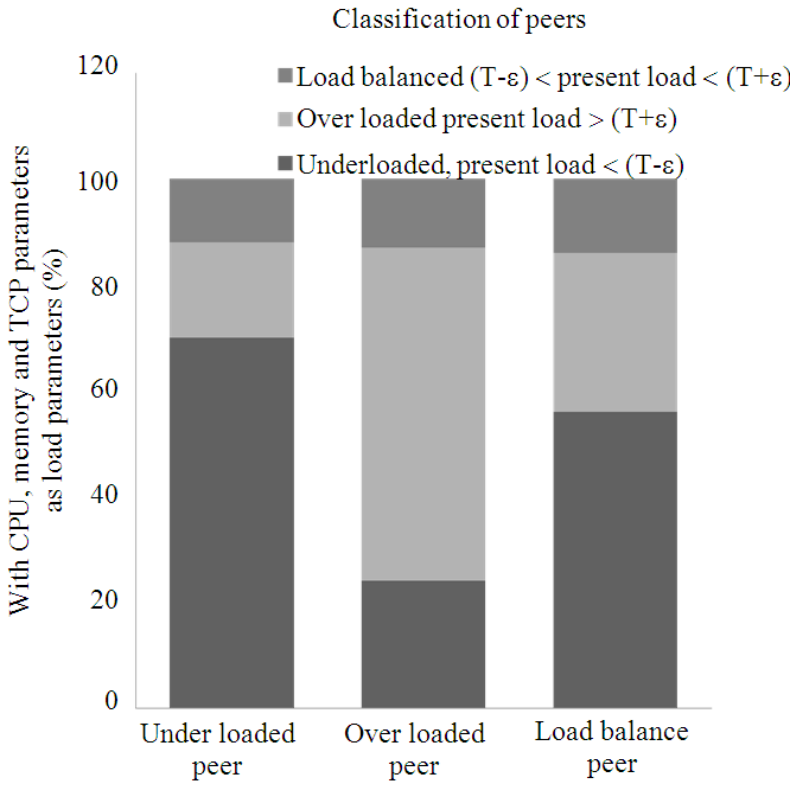

Fig. 2. Classification of peers

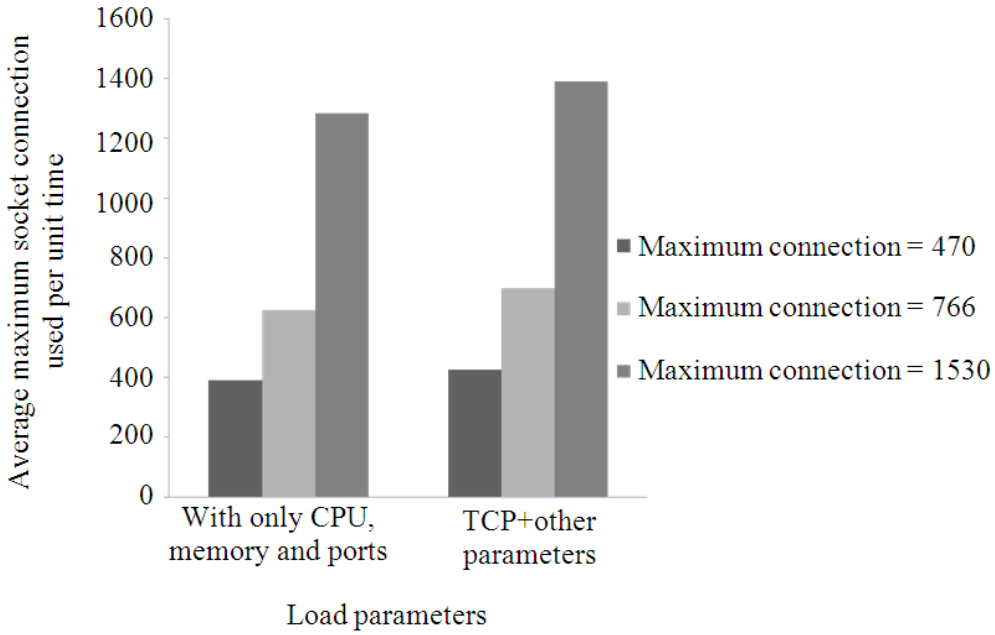

Fig. 3. Number of connections used when load balancing

However, the disadvantage is that the derived load per unit capacity is inaccurate, because it is first calculated locally and then shared with adjacent peers. An alternative scheme is to have the MAs monitoring the peers. All MAs share the sum of loads and capacities of all peers. The MAs calculate the overall load per unit capacity of the entire network using this shared information and inform the peers. This involves more overheads, but the classifications are more accurate because they use almost perfect information to calculate the load per unit capacity. When the peers and virtual servers join and leave, each MA updates the sum of loads of peers and capacities in its monitoring range.

The MAs periodically share their information and the new load per unit capacity is calculated.

7. RESULT AND DISCUSSION

Our proposed algorithm was specially designed to make use of heterogeneity in the load of virtual servers and capacity of peers to reduce the load imbalance factor through exchange mechanism when direct transfer from overloaded to underloaded peer is not possible. Due to this load imbalance factor is kept minimum which is one of the highlight of the proposed algorithm. But in homogenous system suppose every virtual server or task to be transferred is considered as only one unit of load, this algorithm degenerates to 1-N scheme (sender initiated)-shortest selection policy with polling only the under loaded peers indicated by the mobile agent. This happens because when there is no heterogeneity formload function will not be used. In underloaded peer j when a request arrives, if loadj + Li,v ≤ Tj succeeds a response is sent back and if it fails the else part always rejects request in homogenous systems. Though sender initiated scheme suffers during high loads as it can't find underloaded peers thereby increasing the polling activity, our algorithm avoids unnecessary requests during high loads because the mobile agent only gives the details of underloaded peers and so one can classify our scheme stable sender initiated algorithm due to its above feature.

To our knowledge compared to all the existing load balancing algorithms, our proposed work minimizes the load imbalance factor to the greatest extent in heterogeneous systems. This is mainly due to the exchange scheme which is used when one way transfer of virtual server from overloaded to underloaded peer is not possible. When the overloaded peer does not receive any response for one way transfer, it chooses the response from underloaded peer which returns minimum load. When a transfer is not possible, by exchanging the load imbalance factor is reduced for both the overloaded and underloaded peer. Before the exchange of the virtual server, the load balance factor of the overloaded peer is $|T_i - \sum_{v=1}^{k \text{ virtual servers}} L_{i,v}$. The set of virtual servers returned back will always be less than the virtual server moved from overloaded to underloaded peer, i.e., $R_i < L_{i,v}$ Infact the response with minimal Ri is chosen. After the exchange it is reduced to $|T_i - \sum_{v=1}^{k \text{ virtual servers}} L_{i,v} + R_i - L_i|$. Similarly the overall load of the system is increased in receiver peer and thereby decreasing the negative magnitude of $|T_j - \sum_{v=1}^{k \text{ virtual servers}} L_{i,v}$ of the underloaded peer j. Thus it contributes in minimizing the magnitude of load imbalance factor of the underloaded peer.

Figure 4 shows the load imbalance of the system. We determined the number of virtual servers moved for 10 and 100 peers using the simulation discussed in the experimental setup. The proposed load balancing algorithm decides when to migrate a virtual server. We determined the number of virtual servers moved for thousand and ten thousand peers by running the load balancing algorithm in a single program with a P-P distribution. Our proposed schemes of local and global sharing scheme (with inbuilt exchange scheme) has slightly higher load imbalance factor than centralized directory because it uses 1-N scheme. If the same algorithm is implemented as N-N scheme or as centralized directory with exchange mechanism it will do better than existing central directory due to its additional exchange mechanism which will allow more load sharing and thereby minimizing the load imbalance further.

Movement cost includes both the virtual servers which are moved from overloaded peer to underloaded peer and some of the virtual servers which are returned back by underloaded peer from its set of servers. The movement cost is given by:

$$\text{Movement cost} = \sum L_{i,V \text{ moved}+} \sum R_{i \text{ returned back}}$$

The number of virtual servers transferred moved in P-P distribution is shown in **Fig. 5**. Although the movement cost and the number of virtual servers moved is high, the load imbalance of the system is significantly reduced

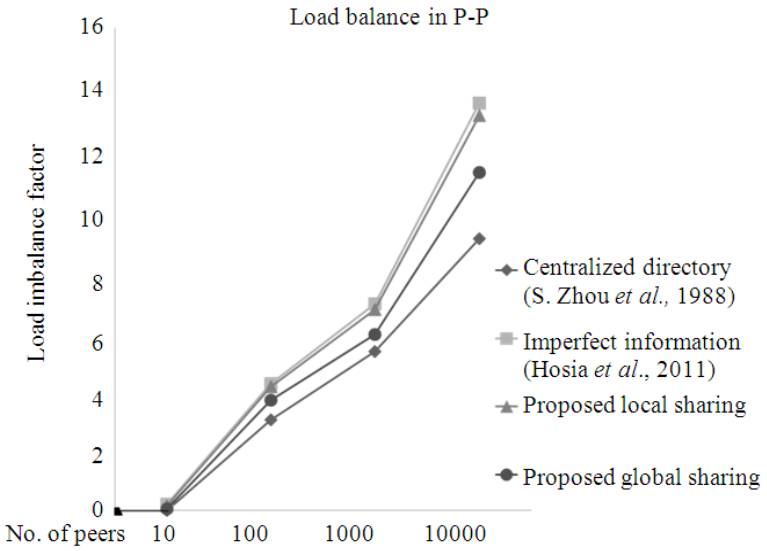

Fig. 4. Load balancing in a P-P distribution

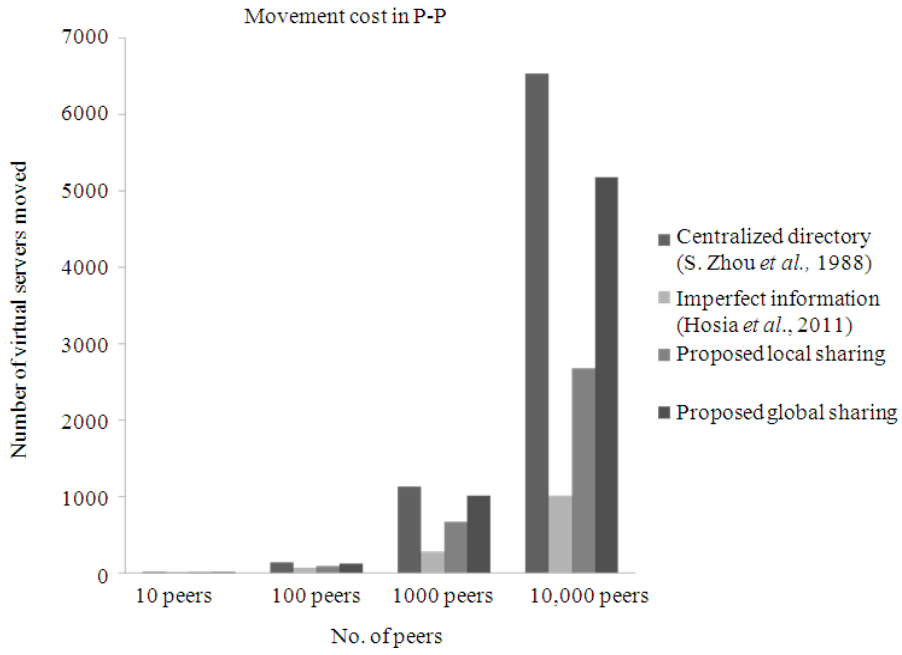

Fig. 5. Number of virtual servers moved in a P-P distribution

8. CONCLUSION

We have presented and discussed a novel load-balancing algorithm that reduces the load imbalance factor. The movement cost due to the transfer of virtual servers is slightly increased, because we have considered reducing the load imbalance to be the primary issue. The method uses mobile agents to calculate the load per unit capacity. Local and global sharing techniques are used by MAs to calculate the load per unit capacity. We

included the maximum possible number of socket connections per unit time as one of the load parameters. Our analysis shows how this improves the classification of the peers into overloaded and underloaded.

If every virtual server or task which is to be transferred is considered as one unit load, the exchange scheme doesn't work at all. Our proposed scheme degenerates to stable sender initiated algorithm with shortest selection policy in this case.

One can consider TCP connections as a significant parameter when there is a considerable CPU and memory capacity. However, we need to address the problem when these resources are scarce. In the future, this 1-N scheme should be converted to a N-N scheme by sharing the responses that are sent by the underloaded peer. After the overloaded peer has selected the best response, the remaining responses can be shared with the other overloaded peers that are monitored by the same MA. This can further reduce the message overheads.

9. REFERENCES

Godfrey, B., K. Lakshminarayanan S. Surana and R. Karp, 2004. Load balancing in dynamic structured P2P systems. Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies, Mar. 7-11, IEEE Xplore Press, pp: 2253-2262. DOI: 10.1109/INFCOM.2004.1354648

Godfrey, P.B. and I. Stoica, 2005. Heterogeneity and load balance in distributed hash tables. Proceedings of the IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies, Mar. 13-17, IEEE Xplore Press, pp: 596-606. DOI: 10.1109/INFCOM.2005.1497926

Hsiao, H.C., H. Liao, S.T. Chen and K.C. Huang, 2011. Load balance with imperfect information in structured peer-to-peer systems. Parallel Distrib. Syst. IEEE Trans., 22: 634-649. DOI: 10.1109/TPDS.2010.105

Karger, D. and M. Ruhl, 2004. Simple efficient load balancing algorithms for peer-to-peer systems. Proceedings of the 16th ACM Symp. Parallelism in Algorithms and Architectures, (AA '04), ACM New York, NY, USA, pp: 36-43. DOI: 10.1145/1007912.1007919

Lamport, L., 1978. Time, clocks and the ordering of events in a distributed system. Commun. ACM,. 21: 558-564. DOI: 10.1145/359545.359563

Ledlie, J. and M. Seltzer, 2005. Distributed, secure load balancing with skew, heterogeneity and churn.

Li, H. and F. Shao, 2011. An improved load balancing algorithm for P2P system based on mobile agent. Proceedings of the 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce, Aug. 8-10, IEEE Xplore Press, Deng Leng, pp: 2791-2794. DOI: 10.1109/AIMSEC.2011.6010282

Li, Z. and G. Xie, 2006. A distributed load balancing algorithm for structured P2P systems. Proceedings of the 11th IEEE Symposium on Computers and Communications, Jun. 26-29, IEEE Xplore Press, pp: 417-422. DOI: 10.1109/ISCC.2006.8

Rao, A., K. Lakshminarayanan, S. Surana, R. Karp and I. Stoica, 2003. Load balancing in structured P2P systems.

Serbu, S., S. Bianchi, P. Kropf and P. Felber, 2007. Dynamic load sharing in peer-to-peer systems. IEEE Int. Comput. Published omput Society, pp: 53-56.

Shen, H. and C.Z Xu, 2007. Locality-aware and churn-resilient load-balancing algorithms in structured peer-to-peer networks. IEEE Trans. Parallel Distributed Syst., 18: 849-862. DOI: 10.1109/TPDS.2007.1040

Vu, Q.H., B.C. Ooi, M. Rinard and K.L. Tan, 2009. Histogram-based global load balancing in structured peer-to-peer systems. Knowl. Data Eng. IEEE Trans., 21: 595-608. DOI: 10.1109/TKDE.2008.182

Wang, J. and M. Vanninen, 2006. Self-configuration protocols for P2P networks. Web Intell. Agent Syst., 4: 61-76.

Wang, X., Y. Zhang, X. Li and D. Loguinov, 2004. On zone-balancing of peer-to-peer networks: Analysis of random node join. Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, Jun. 12-16, ACM New York, NY, USA., pp: 211-222. DOI: 10.1145/1005686.1005713

Zhu, Y. and Y. Hu, 2005. Efficient proximity-aware load balancing for DHT based P2P systems. IEEE Trans. Parallel Distributed Syst., 16: 349-361. DOI: 10.1109/TPDS.2005.46

Zou, L., E.W. Zegura and M.H. Ammar, 2002. The effect of peer selection and buffering strategies on the performance of peer-to-peer file sharing systems. Proceedings of the l0th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Oct. 16-16, IEEE Xplore Press, pp: 63-70. DOI: 10.1109/MASCOT.2002.1167061

Nehra, N., R.B. Patel and V.K. Bhat, 2007. A framework for distributed dynamic load balancing in heterogeneous cluster. J. Comput. Sci., 3: 14-24. DOI: 10.3844/jcssp.2005.323.331