ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology
Volume 1, Issue 3, May2012

SEIZETOKEN: A DYNAMIC LOAD BALANCING ALGORITHM FOR DISTRIBUTED SYSTEMS

[1]Ankita Singhal, [2]Padam Kumar
[1] M.Tech IIT Roorkee INDIA,[2]Head of Department IIT Roorkee,INDIA,

Abstract: **With the increase in the number of concurrent users on the Internet, the load balancing problem in distributed systems is becoming more significant.To the best of our knowledge, almost all strategies take the support of broadcasting of load information and a lot of task transfer if a task reaches a heavily loaded server, or in some cases, a node which is just heavier than the other node. Broadcasting of load information increases significant traffic congestion on the network and also makes the load information stale. And task transfer sometimes takes time more than the service time of a task. This paper gives a load balancing strategy named SeizeToken which eliminates any need of load information. Also it tries to minimize the need of task transfer up to a large extent. The algorithm proposed is inspired by the Token ring algorithm in computer networks. Seize Token is compared with some existing load balancing strategies. The simulation results show that SeizeToken gets a much better response time than others. Also some variations of SeizeToken has been compared.**

Index terms: **Distributed systems, heterogeneous, load balancing, neighboring nodes, random, semi distributed, token**

I. INTRODUCTION

A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility [1]. This definition considers a system to be distributed only if the existence of autonomous nodes is *transparent* to the users of the system. A system distributed in this sense behaves like a standalone computer system, but the implementation of this transparency requires the development of intricate distributed control algorithms.

In distributed system, load balancing services distribute client workload among various back-end servers in order to obtain the best response time possible. Moreover, it should be avoided that some tasks are forced to wait for a very long time. Typically any load balancing strategy tries to select a lightly loaded node and transfer the load to this node. That is, the load information of nodes is used. However, using load information is not so easy. The information gets old due to the communication delay (if probed on demand) or up to the update interval (if updated periodically) [2]. Old load information does not reflect the current load of the system. There are some researches to deal with the effect of old load information in task allocation [2], [3], [4]. To suppress the effect of old information these approaches mainly introduce some randomness to break the symmetry. So, taking this point we use a purely random mechanism to select a particular group of nodes. Our simulation results reveal that for small systems, having tasks go to the least loaded server can significantly hurt performance. Using random approach enhances the overall performance of the system.

The main aim of most of the existing strategies for load balancing is that each node must have almost equal load distribution i.e. if there are 4 nodes in a system then each must have near to 25% of total load. But in an attempt to distribute the load in such a disciplined manner, the main aim of improving the response time of the client is overlooked. The complexity of the algorithms leads to degradation of response time rather than improving it. Till a server has tasks much less than its capacity it should readily accept more tasks to run without bothering about the load on other servers. This will save the time of communication with other nodes, exchanging load information with them and then finally deciding where the task will be run. If a server is under loaded, it must execute a task immediately

II. PROBLEM STATEMENT

The existing strategies for task allocation mainly depend on load information for choosing the node to which a task can be allocated. More specifically, the task allocation scheme called Nearest Neighbor (NN) proposed in [5] uses load information of nodes. In NN, each node stored information of all of its neighbors. If the load of a node is changed i.e. a new task is added to the run queue of the node or a task finished executing, the node sends a message to *all* of its neighbors immediately. Receiving the message, the neighbors update its load information. Now when a new task arrives, the local node always determines whether the task can be executed locally or should be transferred. If the destination node is more appropriate than local node, the task is transferred. Otherwise the task is executed locally.

There are four main problems that can be highlighted in the above explained strategy. The problems may not be independent and seek solutions in one another.

- Broadcast of load information

Distributed dynamic load balancing algorithms tend to generate more messages than non-distributed algorithms. This is due to the fact that each node might need to interact with all other nodes in the system in order to make its load balancing decisions. Although the strategy proposed in [5] does not interact with all of the nodes in the system but only with its neighbors, but since the load over any node is subject to frequent changes, a lot of bandwidth is consumed in sending and receiving the 'load information messages' even with the neighbors.

- Only for homogeneous distributed systems

Another problem that rises from this broadcasting of load information is that all the nodes in the system must decide on a common parameter that will decide its load. But in a heterogeneous system, there may be varying speeds of processors of nodes which will have different parameters for load.

- Task transfer – a complex operation

Task transfer is not a simple operation and involves great deal of overhead. This increases processing delay.

- Old load information

The last problem is the way load information is updated. The node sends a message to all of its neighbors immediately after the load of a node is changed. Updating load information only with neighboring nodes only alleviates the old information problem and does not eliminate it. That is, when node whose load is changed, sends its load to neighboring nodes, the load of that node may have again been changed till the message reaches neighboring node [3].

In this paper, we propose a solution that aim at minimizing the broadcast of load information and also minimize the need for task transfer up to a great extent thus decreasing the processing delay involved. The strategy proposed is such that the old load information problem is completely eliminated. Also, the proposed solution does not bound to just homogeneous distributed systems but can also be implemented for the heterogeneous distributed systems.

III. PROPOSED APPROACH

The gaps that are just discussed, if noted carefully, are mainly because the system is implemented using a pure distributed approach. Since they generate more messages and so exchange them a lot.

But using a purely non distributed approach will also not serve the purpose since it lacks fault tolerance. Centralized algorithms jeopardize system performance in the event that the central node crashes. Also, there is a possibility that this node could cause a bottleneck if it became swamped with messages from all the other nodes in the system. On the other hand, balancing requires fewer messages to reach a load balancing decision. This is because other nodes in the system do not interact with each other; they only interact with the central node. A study by [6] has shown that centralized load balancing suits small sized networks (less than 100 nodes) more than any other control method.

Taking the advantages of both type of approaches and eliminating the disadvantages, here we use a semi distributed system. In a semi-distributed form, e.g., [7], nodes of the distributed system are segmented into clusters. Load balancing within each cluster is centralized; a central node is nominated to take charge of load balancing within this cluster. Load balancing of the whole distributed system is achieved through the cooperation of the central nodes of each cluster

In NN, the load information is sent only to neighboring nodes. The neighboring nodes were chosen to suppress the effect of old load information [2], [3] since communication delay between the neighbors is short. But what defines a neighbor? The paper remains silent on exactly which nodes must be considered neighbors or which nodes collectively form a group.

The proposed solution is divided into two major modules:

i. Algorithm for dividing the network into groups of neighboring nodes

ii. A protocol that will be used by each group for task allocation with minimum broadcasting and processing delay.

The first module follows a graph theoretic approach whereas the second module follows the heuristic approach.

A. Algorithm to divide network into groups

In this section, we will describe the algorithm for dividing the network into groups of neighboring nodes. Actually if the network is looked as a graph with nodes as vertices and the interconnections between the nodes as edges then the problem of finding neighbors decomposes into finding all cliques of a graph [8] , [9]. The Bron Kerbosch Algorithm [9] of finding cliques is the mostly used. But the complexity of such algorithms is very high. Finding all cliques is a NP Complete Problem.

Thus, the algorithm that is proposed here does not use any standard neighbor finding algorithm. The algorithm is kept simple which just meets the needs of our load balancing algorithm (explained in next section). As discussed earlier, we want to suppress the effect of old load information. For this, it is necessary that the task transfer takes place only with the neighboring nodes since the communication delay between the neighbors is short. Secondly, here we use a centralized approach to task allocation (explained later). The centralized task allocation schemes are originally proposed for small systems. It is difficult to implement them to large scale distributed systems due to lack of fault tolerance. Thus we must have minimum number of groups. Keeping these points in mind, the algorithm proceeds as follows:

Given a network of nodes, find out the node with the highest degree. This will be the MasterNode of the group. Put all the nodes attached with this node in one group. Repeat the process (without considering the nodes that have already become a part of any of the groups) until each node becomes the part of any of the group.

If some groups contain just a single node do the following: Find out nodes that were originally connected to this single node. Take any random node from this set. Call it subMasterNode. Make the single node part of the same group of which subMasterNode is a part and attach it to the subMasterNode. Remove the single node group. This is done to keep the number of groups as small as possible.

Algorithm for the above explained solution is as follows:

V is the set of nodes in the system.

P is the set of nodes that have not been a part of any of the groups.

M is the set of Master nodes; SM is the set of subMasterNodes

N is the set of sets of nodes that are a part of the MasterServer group.

SN is the set of sets of nodes that are part of SubMaster nodes.

```
1.  Initialize P = V; N = null, M = null, SM = null

2.  Repeat steps 3 to 5 until P becomes empty

3.  Find the highest degree node say MasterNode from set
    of nodes contained in P. Add MasterNode to M.

4.  Find the set of nodes from set P adjacent to
    MasterNode. Also add MasterNode to this set and add
    it to set N. These will be the members of the group
    whose leader will be MasterNode.

5.  Update P i.e. remove all the nodes found in steps 3 and
    4 from P

6.  Minimize (N)

7.  Exit
```

Minimize (N)

```
1.  Initialize T = N; SN = null

2.  Repeat steps 3 to 10 until T becomes empty

3   .Let s be a set from T. If s contains a single node say n go
    to step 5

4.  Else go to step 10

5.  Find the set of nodes from set V adjacent to n.

6.  Take any random node from the set found in step 5.
    Call it subMasterNode. Add this node to set SM. Also
    add this node to SN

7.  Find out the group of which subMasterNode is a part.

8.  Make n part of the group found in step 7. Also add set
    containing s and subMasterNode to SN

9.  Update N:

        Remove s from N

    Update M:

        Remove s from M

10. Remove s from T

11. Exit
```

In the above algorithm, Minimize (N) ensures small number of groups. Also, here we make groups only with the nodes that are adjacent to MasterNode. That is, these nodes will be directly connected to MasterNode and thus will have short communication delay.

The algorithm is run with an example taking the network of Fig 1.

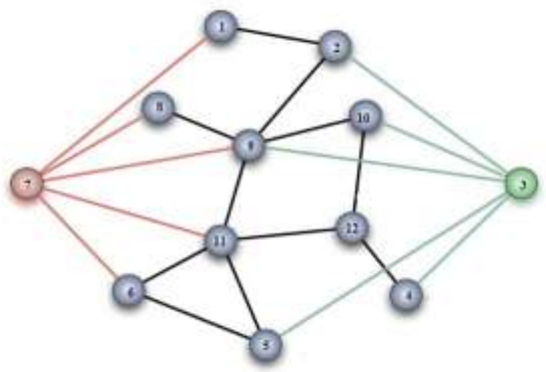

Fig 1: Initial Network graph

deg(x) defines the degree of the node x.

The Algorithm for Fig 1 works as follows:

V = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

Initialize:

P = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

N = M = SM = null

Iteration 1:

Fig 1 shows the graph for iteration 1

deg(1) = 2, deg(4) = 2, deg(7) = 5, deg(10) = 3, deg(2) = 3 deg(5) = 3, deg(8) = 3, deg(11) = 5, deg(3) = 5, deg(6) = 3, deg(9) = 6, deg(12) = 3

The highest degree is of node 9. Thus, MasterNode = 9

M = {9}

Set of nodes adjacent to node 9 = {2, 3, 7, 8, 10, 11}

Adding MasterNode to this set we get {9, 2, 3, 7, 8, 10, 11}

Thus, N = {{9, 2, 3, 7, 8, 10, 11}}

P = {1, 4, 5, 6, 12}

Iteration 2:

Fig 2 shows the graph for iteration 2

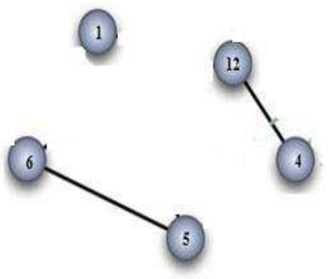

Fig 2: Graph for iteration 2

deg(1) = 0, deg(6) = 1, deg(4) = 1, deg(12) = 1, deg(5) = 1

There are four highest degree nodes 4, 5, 6 and 12. Randomly, taking 4 as highest degree node we have,

MasterNode = 4

M = {9, 4}

Set of nodes adjacent to node 4 = {12}

Adding MasterNode to this set we get {4, 12}

Thus, N = {{9, 2, 3, 7, 8, 10, 11}, {4, 12}}

P = {1, 5, 6}

Iteration 3:

Graph for iteration 3 is shown in Fig 3

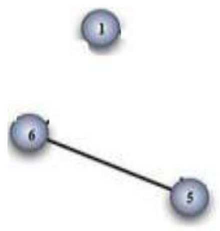

Fig 3: Graph for iteration 3

deg(1) = 0, deg(5) = 1, deg(6) = 1

There are two highest degree nodes 5 and 6. Randomly, taking 6 as highest degree node we have,

MasterNode = 6

M = {9, 4, 6}

Set of nodes adjacent to node 6 = {5}

Adding MasterNode to this set we get {6, 5}

Thus, N = {{9, 2, 3, 7, 8, 10, 11}, {4, 12}, {6, 5}}

P = {1}

Iteration 4:

Fig 4: Graph for iteration 4

Graph for iteration 4 is shown in Fig 4

deg(1) = 0

The highest degree is of node 1. Thus,

MasterNode = 1

M = {9, 4, 6, 1}

Set of nodes adjacent to node 1 = { }

Adding MasterNode to this set we get {1}

Thus, N = {{9, 2, 3, 7, 8, 10, 11}, {4, 12}, {6, 5}, {1}}

P = { }

Since P becomes null, we now run Minimize (N)

T = N = {{9, 2, 3, 7, 8, 10, 11}, {4, 12}, {6, 5}, {1}}

Minimize (N)

Iteration 1:

s = {9, 2, 3, 7, 8, 10, 11}

s does not contain a single node. Thus removing s from T

T = {{4, 12}, {6, 5}, {1}}

Iteration 2:

s = {4, 12}

s does not contain single node. Thus removing s from T

T = {{6, 5}, {1}}

Iteration 3:

s = {6, 5}

s does not contain single node. Thus removing s from T

T = {1}

Iteration 4

s = {1}

s contain single node.

Set of nodes adjacent to node 1 (from Fig 1) = {7, 2}

Randomly taking node 2 as subMasterNode and adding node 1 to the group of which subMasterNode i.e. node 2 is a part. Since node 2 is a part of group {9, 2, 3, 7, 8, 10, 11}, adding 1 to this and removing s from N we have N updated to

SM = {2}

N = {{9, 2, 3, 7, 8, 10, 11, 1}, {4, 12}, {6, 5}}

M = {9, 4, 6}

SN = {2, 1}

T = { }

Stop since T becomes null.

Thus, Master nodes = {9, 4, 6}

SubMasterNode = {2}

Groups = {{9, 2, 3, 7, 8, 10, 11, 1}, {4, 12} {6, 5}}

Subgroups = {{2, 1}}

The final groups formed are shown in Fig 4.5

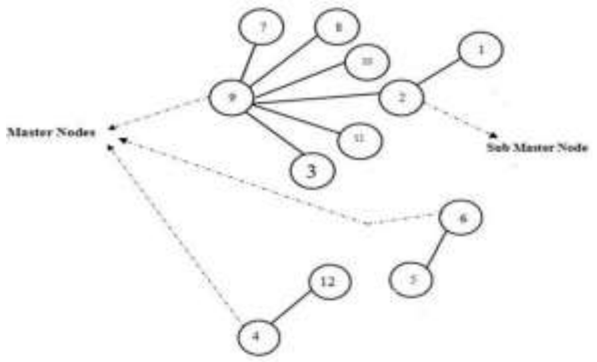

Fig 5: Groups after running the algorithm

B. SeizeToken Algorithm

In this section, we will describe the algorithm that will be used for balancing load inside each group. This will be a distributed approach. We will call this approach as SeizeToken. First we divide the nodes into various groups using the algorithm described in previous section. Now, within each group there is a MasterNode, a node that is connected to all other nodes in the group, there may be a subMasterNode, a node that is directly connected to MasterNode and also connects two or more nodes. Also, there will be an interface between the client and the groups, which will simply forward the requests coming from clients to any of the randomly selected group. We call this interface as pseudoRouter. So when a task arrives, pseudoRouter randomly selects a group and forwards the request to the MasterNode of the selected group. The MasterNode then checks if the task can be executed locally, if not, it generates the token, broadcasts it to all the members of the group (except the nodes connected to subMasterNode). As soon as a task arrives, any of the servers that is ready to accept the request (that is length of run queue is less than its capacity) seizes the token. Once the token is seized no other node can seize it, since the token is unique in a group. Thus, the task is completed by the node that seized the token and then destroys it. Each node must notify the Master Server if they become full (length of the run queue reaches maximum limit) since at any point of time, if, all the nodes of a group become full then the Master Server may not accept any further request and may notify this to pseudoRouter.

The architecture for the proposed solution is shown in Fig 6.

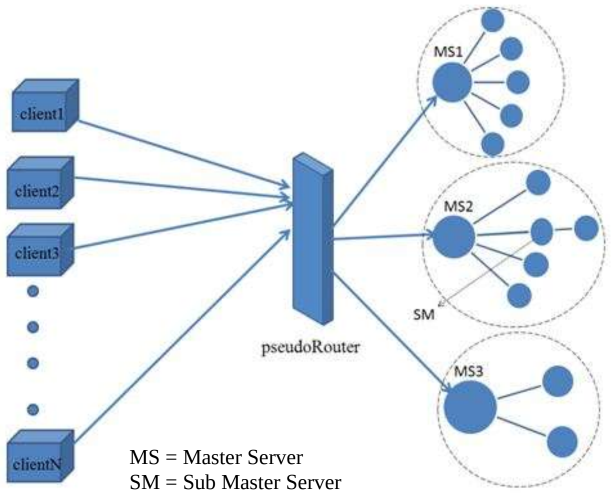

Fig 6: Architecture of the proposed work

SeizeToken Algorithm is as follows

token : a sharable variable among all the nodes of a group

maxCapacity : the maximum number of requests that a node can handle

generateToken() : generates a long random sequence of bits

broadcastToken() : forwards the value of token to all the nodes directly attached to it

SeizeToken() : inverts the first bit of the token thus making it dirty so that other nodes cannot access it

acceptRequest(r) : puts r in queue and increments the value of queueLength by 1;

notSeized () : Checks if the token is seized or not

acceptToken () : Saves the value of the token broadcast by MasterServer

```
MasterNode () {

if (queueLength>=maxCapacity)

//if master server is overloaded generate token so that other servers of the group can execute the request

{

    generateToken ();

    broadcastToken ();

 }

  else

    acceptRequest (r);

}
```

```
subMasterNode ()

{

  acceptToken();

//accept the token that was broadcast by MasterServer

  if (queueLength>=maxCapacity)

 // if submaster server capacity is full, broadcast token to the nodes attached to this node

    broadcastToken();

 else {

    if (notSeized())

//check if token has not yet been seized by some other node

    {

      SeizeToken ();

       acceptRequest(r);

  }}}
```

```
otherNodes() {

acceptToken();

 if ((queueLength<maxCapacity)&(notSeized()))

//check if the capacity of this node is not full and token has not been seized by any other server

{

       SeizeToken ();

       acceptRequest(r);}}
```

The main point to be noted here is that, the only communication among the nodes is between the MasterNode and its neighbors. Thus broadcasting is done only between the directly connected nodes. Token is broadcast instead of load information. Thus, there is no danger of getting load information stale. The balancing does not depend on any load information or any specific parameter. Every individual node can take its own decision without bothering about the load of other nodes in a group or system.

C. Fault Tolerance in SeizeToken

The main feature of distributed systems is its fault tolerance capacity by having replicated nodes. The architecture proposed has a centralized entity called the pseudoRouter. In any network, a router will be connected to every node. Thus every response to any request will go through it. Now if pseudoRouter fails, even then the request will be forwarded to any randomly selected node by the network router.

Also, if Master Server of any of the groups fail, then there are two methods to handle this. First, notify the failure of Master Server to pseudoRouter and then pseudoRouter will make an entry in its log so that while selecting the group it does not select tthe group of which the failed node is the master server. Second, run the algorithm to divide the network into groups, again, without considering the links of the failed node. The main advantage of using the first method is its simplicity and low cost. But a major drawback in it is that all the nodes attached to the failed master server node will also be considered failed since none of the request will reach them . So this method will lead to failure of a group in case the master server fails. This will also increase the load on the remaining active groups. So the second method, though a bit complex, makes more sense. That is, as soon as the master server fails we run the algorithm again taking the original network into consideration. This will affect only the failed node and not other nodes. New groups will contain all active nodes.

On the other hand, if any of the sub master servers or any other nodes fail, we will use the first method i.e. we will not consider that node at all and no more requests will be sent to that node. This will be notified to master server so that while broadcasting token, it will not broadcast to the failed node. Again, in case of sub master server failure, it will lead to loss of the nodes that are attached to it, but since very less number of nodes are normally the part of sub master server, so it will not affect the performance much. But running the algorithm again for sub master server may make it too complex and does not worth it.

IV. IMPLEMENTATION DETAILS AND EXPERIMENTAL RESULTS

We consider distributed systems consist of many nodes. Nodes can communicate with each other. Internodes' communication involves some communication delay. Each node can process only one task at a time and an executing task cannot be interrupted. Tasks are processed in FCFS manner. Each node has a run queue. Tasks that are allocated to a node but cannot be executed immediately are put in the run queue of the node. The load of a node is defined as the length of its run queue, i.e. the number of tasks in the queue. The mean response time is defined as the time from when the request is made to the time when the request is served back to client. In order to determine the behavior of the scheme mentioned in Chapter 4, I performed the following simulations:

Assumptions:

- All tasks are independent, that is, execution of a task at a node has no influence to other tasks.
- All the requests that arrive have same service time

We took a network shown in Fig 1. Divide it into groups using the proposed algorithm. Finally the groups that were made are shown in Fig 5. The nodes in any particular group are neighbors. Then Nearest Neighbor was implemented inside each group i.e. each node stores load information of all its neighbors. When load of any node changes, it sends a message to all its neighbors immediately. Now when a request arrives, the local node determines the destination node. If the destination node is more appropriate, the task is transferred. SeizeToken is also implemented using the same network of Fig 5. Fig 7 shows the variation in response time of both the algorithms.

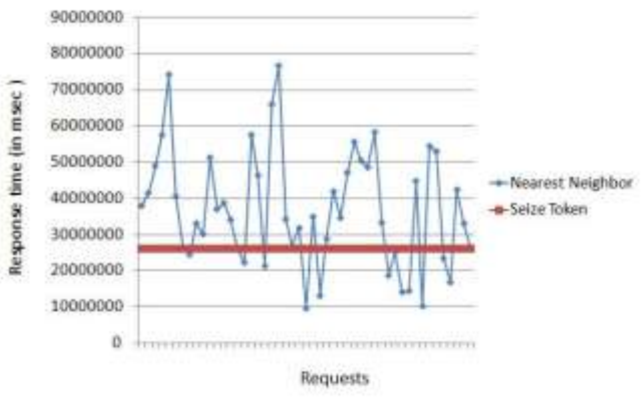

Fig 7: Comparison of response times of Nearest Neighbor and the proposed algorithm SeizeToken

The response time is calculated for 50 concurrent requests. Response time of NN is non monotonous in nature since each request is sometimes executed from there itself and sometimes transferred. The extreme peaks show that the request is transferred multiple times from one node to another. At the extreme, a form of 'processor thrashing' may occur, in which all of the nodes are spending all of their time transferring tasks. Since task transfer is a complex operation thus depending on number of times the request is transferred, the response time increases or decreases. The extreme lower values usually show that the request is served from the local node. As can be seen, sometimes, serving from the local node gives better response time than SeizeToken. But such requests are very less.

SeizeToken, on the other hand, appears to be a constant graph. Actually it is not exactly constant but increases at a very slow rate.This is because as the load on any server increases the response time tends to increase. But this increase is very small as compared to task transfer mechanism in NN. Overall, Fig 7 shows that the algorithm proposed SeizeToken, gives a much better response time than the existing Nearest Neighbor Algorithm.

The performance of random location strategy, which does not employ any information in its selection, was significant as compared to the load balancing algorithm that collect global information. Fig 8 proves this clearly. The Seize Token algorithm is implemented with 3 variations:

Least Congested, Round Robin, Random

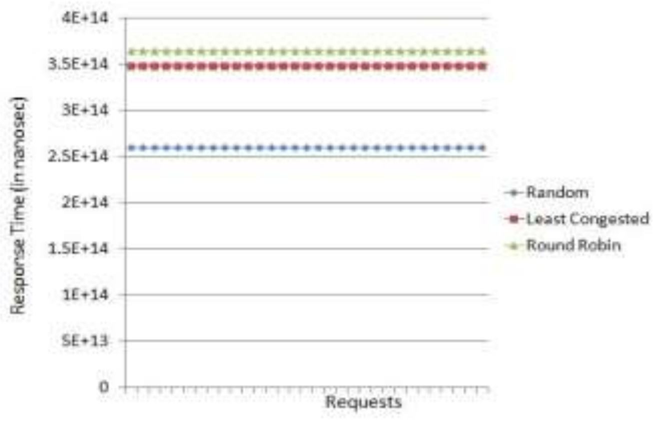

Fig 8 Comparison of response times with Least Congested, Random and Round Robin techniques.

Least Congested

In this implementation, when request arrives and reaches the pseudoRouter, it first collects the load from each group. The load of a group is defined as the sum of the load of all the nodes in a group. Having collected the load from each group it sends the request to the group with minimum load. Doing all this increases the complexity of the pseudoRouter and also the overall response time as shown in Fig 8.

Round Robin

In this implementation, when first request arrives it is sent to the 1st group of the system. The second request is sent to the 2nd group and so on in a round robin fashion. The load on each group obviously is distributed in a very balanced manner but the simulation results shows that in terms of response time it gives the worst performance. The response time calculated in this case is even more than Least Congested

This is the algorithm that is proposed. In this when a request arrives, pseudoRouter simply forwards it to any randomly selected group. Since this is the least complex implementation, the response time obtained in this case is the best. Thus our Seize Token uses this method of selecting a group of nodes.

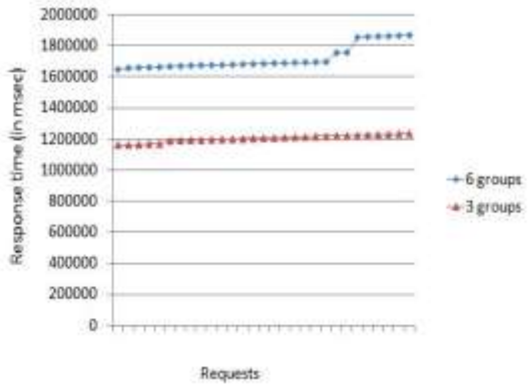

Fig 9 Comparison of response times for different number of groups

Fig 9 shows the need for keeping minimum number of groups. To obtain the above results, a network of 19 nodes was taken and divided it into 6 groups (not following the algorithm proposed). The response time was calculated and plotted for this case. Then for the same network, the proposed algorithm for dividing the network was applied. It divided the network into 3 groups. Again the response time was calculated and plotted.

The results clearly show that for the same number of nodes small number of groups gives better performance.

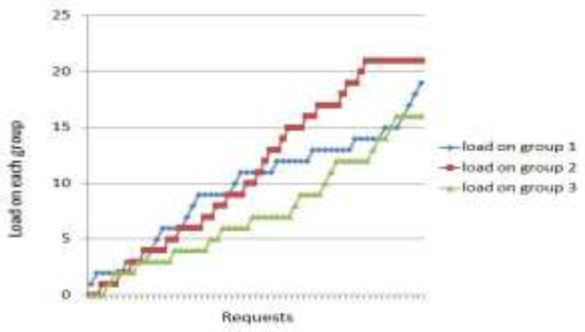

Fig 10 Comparison of load on each group in Random variation of SeizeTokenAlgorithm

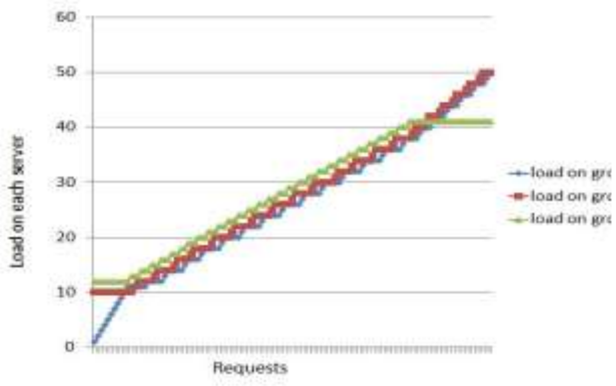

Fig 11 Comparison of load on each group in Least Congested variation of SeizeToken

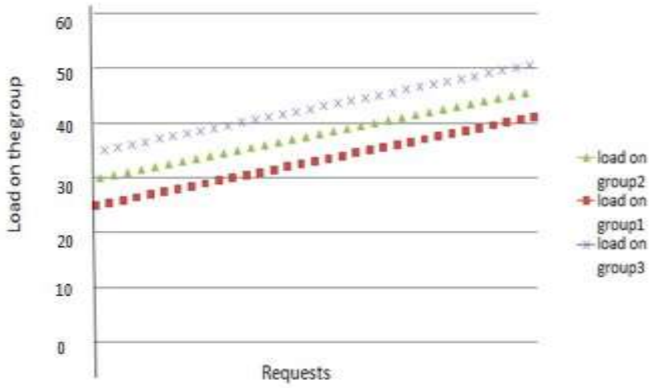

Fig 12 Comparison of load on each group in Round Robin variation of SeizeToken

Fig 10 results are obtained for a network that is divided into groups. For 50 concurrent requests the load was plotted for each group. The load of a group, as defined earlier, is the sum of the load of all the nodes in a group. As can be seen, due to randomness involved in selecting a group, group 3 remains the least loaded group throughout the simulation, whereas group1 and group2 have comparable load up to a certain time and after that group 2 becomes the most heavily loaded server.

Fig 11 is for Least Congested version of Seize Token i.e. when request arrives and reaches the pseudoRouter it first collects the load from each of the groups. Having collected the load from each group it sends the request to the group with minimum load. The results are obtained for the network that is divided into 3 groups. The initial load on each group is kept different i.e. at the time of simulation, group1 was least loaded and group 2 and group 3 had almost same load. Now, initially, all the requests are served from group 1 since it is the least loaded group. But when group 1 also becomes comparable to group 2 and 3, then requests are transferred to group2 and group 3. Also, since group 2 and group3 have approximately same load the requests goes to them approximately alternately. Thus, a knot like graph is obtained for this case. But when group2 reaches its maximum capacity, group2 serves all the requests. This condition mainly occurs when the load between 2 groups differs just by a unit or two.

Fig 12 shows the variation of load on each group in case of Round Robin. For clarity, the intermediate joining lines are removed. The initial load on each group was kept different. But irrespective of this, Round Robin distributed load one by one on each request. Thus, the load on each group increases in a much disciplined manner.

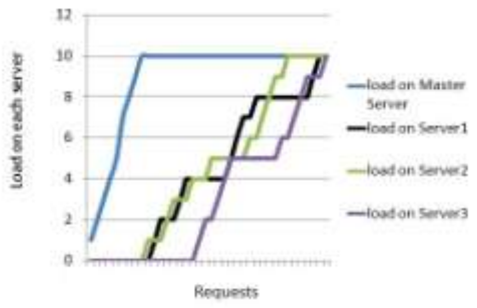

Fig 13 Comparison of load on each node in SeizeToken

Fig 13 shows the compromise that is done to avoid task transfers. The results are obtained for a group with 3 nodes and a Master Server. It gives an overview of load on each server at any instant of time. The load distribution in SeizeToken is not very even as same server can seize token again and again to execute a request. Also, since according to the algorithm, Master Server serves the request till its maximum capacity and then forwards it, initially its load becomes very high and then remains constant throughout. The load distribution was found to be more disciplined in case of NN. But this does not affect the response time much and only improves it over NearestNeighbor.

V. CONCLUSION AND FUTURE WORK

A. Conclusion

The work proposed gives a solution to all the problems discussed in section II. The following conclusions can be made:

- The broadcasting of load information is completely eliminated. Instead, a token, which is just a sequence of bits, is broadcast. The token used here is much lighter than the load information packets that are sent through the networks. Thus negligible traffic is generated on network by the broadcast of token as compared to load information messages

- The interaction among the nodes is minimum since for those algorithms that require each node to exchange status information with every other node in the network, distributed control could be a great burden on the communication system which affects the overall system performance negatively. More specifically, the only interaction is among the nodes that act as Master server and the nodes attached to it.

- The architecture proposed is well suited for heterogeneous systems since there is no specific parameter of load on basis of which load balancing is performed. Though in our simulation we used a common parameter (of number of tasks in run queue) as load for all the nodes, but the proposed algorithm is completely independent of the type of load, since its solely the decision of an individual node whether to accept an arriving task or not. This can be seen as a significant feature in heterogeneous distributed systems where there are processors of varying speeds and thus, will have different parameters for defining load.

- The algorithm proposed follows 'if I can serve then only I will accept'. That is, no matter whether a neighboring node is lightly loaded or heavily loaded, a node will accept the request if and only if it can serve it efficiently otherwise it will not. The main advantage of this policy is the minimal task transfer since task transfer is a very complex operation. Also, if a node would have searched for some lightly loaded node, then a kind of processor thrashing may occur where nodes spend more time transferring tasks than executing it.

- The problem of old load information is completely eliminated since the decision whether to accept the request or not is made at the moment the request arrives. There is no load information exchange, and all the decisions are based on the freshest information of the system.

B. Future Work

- The work can further be extended by using some parameter to choose a particular group rather than using a purely random mechanism. But this should obviously prove better than random technique used.

- In future, some more work can be done so that even the token is not broadcast i.e. broadcasting is eliminated completely.

- In case of workload bursts all of the nodes may become overloaded and may reject the arriving requests. A buffer or a new queue may be formed which will keep these requests and will serve them when the nodes become free.

REFERENCES

[1] A. Tanenbaum and M.V. Steen. "*Distributed Systems: Principles and Paradigms*", Prentice Hall, Pearson Education, USA, 2002.

[2] M. Mitzenmacher. "How useful is old information?," *IEEE Transactions on Parallel and Distributed Systems*, vol.11, no.1, pp.6-20, January 2000.

[3] M. Dahlin. "Interpreting stale load information," *In Proceedings of 19th IEEE International Conference on Distributed Computing Systems*, pp.285-296, Austin, USA, Jun 1999.

[4] A Deng, Y. A Lau, R.W.H. "On Delay Adjustment for Dynamic Load Balancing in Distributed Virtual Environments" *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 4, pp. 529-537, April 2012

[5] H. Tada. "Nearest Neighbor Task Allocation for Large-Scale Distributed Systems", *10th International Symposium Autonomous Decentralized Systems* (ISADS), pp. 227-232, Japan, March 2011.

[6] S. Zhou. "A Trace-Driven Simulation Study of Dynamic Load Balancing", *IEEE Transactions on Software Engineering*, vol. SE-14, no. 9, pp. 1327-1341, September 1988.

[7] I. Ahmed and A. Ghafoor. "Semi-Distributed Load Balancing for Massively Parallel Multicomputers," *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp 987-1004, October 1991.

[8] H. Ishii and H. Kakugawa. "A self-stabilizing algorithm for finding cliques in distributed systems", *In proceedings of 21st IEEE Symposium on Reliable Distributed Systems*, pp. 390-395, Japan, 2002.

[9] C. Bron and J. Kerbosch. "Algorithm 457: finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, no. 9, pp. 575-577, 1973.