ISSN No. 0976-5697

Volume 2, No. 4, July-August 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

A Semi-Distributed Approach for Dynamic Load Distribution in Distributed Systems

Sandipan Basu
Post Graduate Department of Computer Science
St. Xavier's College
Kolkata, INDIA.
mail.sandipan@gmail.com

Abstract: This paper presents an efficient algorithm, which is able to distribute the workload dynamically in a distributed systems. In this algorithm, we do process allocation in such a way, that resource utilization will be robust and also managing the workload among different nodes (processors) and also kept a look on load sharing, so that no node will become (almost) idle. In this algorithm, I suggest a semi-distributed structure of nodes, so that certain problems (that usually appear in case of load balancing & load sharing in distributed system) can be solved without much effort.

Keywords: dynamic load sharing, semi-distributed, local process, remote process, mcount, local table, global table, local coordinator, cluster, CPU utilization.

I. INTRODUCTION

In distributed systems, process management is a significant issue, since processes are distributed among several nodes (processors) throughout the system, and to manage those processes within processors need some serious management. By the term load distribution, we mean, load sharing and load balancing along with process migration [7]. Load balancing in distributed system, refers to the distribution of workload among several nodes, so that the workload is (approximately) evenly balanced throughout the system. But, in the real & hard sense, absolute load balancing in distributed system is not possible, since the number of processes in a node is always varying and there exists a temporal unbalance among the nodes at every moment. Hence, the aim is to keep a moderate balance of workload between nodes, as far as possible, with minimum effort. But in case of distributed system, besides load balancing, our main aim is that, every node must perform some reasonable amount of work, so that no node becomes (almost) idle, especially, when one node becomes heavily loaded. Hence, in our approach, we also discuss the concept of dynamic load sharing.

In the proposed algorithm, we use the concept of semi-distributed system. That is, we consider a system that is neither pure centralized, nor pure distributed; we call it as *semi-distributed*. With this new kind of approach, what we try to do is, to remove the problem associated with pure centralized system and pure distributed system. Here, we use the term, ***local process*** – a local process is the one that is processed at its originating node; and ***remote process*** – a remote process is one that is processed at a node, different from the one on which it originated [2].

II. PRECONDITIONS

The effectiveness of algorithm depends on the validity of assumptions that are made. The following assumptions are needed to make the algorithm work appropriately.

a) All nodes in the entire system are assigned identification number say, 1 to N.
b) All nodes in the system are fully connected.

As we are considering distributed systems, some assumptions also need to make about the communications network. This is very important because nodes communicate only by exchanging messages between them. The following aspects about the communications network should be considered.

a) Messages are not lost or altered and are correctly delivered to their destination in a finite amount of time.
b) Messages reach their destination in a finite amount of time, but the time of arrival is variable.

III. ALGORITHM

As mentioned above, in the proposed algorithm, we consider semi-distributed system. We define semi-distributed system as a system, which is neither purely centralized nor purely distributed; it is in an intermediate form. In the semi-distributed system, if there are *n* nodes (processors), then out of them, $floor(\sqrt{n})$ nodes are selected as *local coordinator(s).* A task of local coordinator is to take care (meet the requirements needed for load distribution) of **<u>at the most</u>** ($floor[n \div floor(\sqrt{n})] - 1$) nodes. A local coordinator with its associated nodes forms a *cluster.* The entire distributed system is divided into multiple clusters, where each cluster is (primarily) responsible for dynamic load distribution for its associated nodes. Again, local coordinators are inter-connected and exchange information as needed. In the proposed algorithm, we follow certain policies, which are suitable for the algorithm.
These are –

A. *Priority Assignment policy :- Intermediate*

Whenever a process is generated or fetched into a system, a priority has been assigned to that process. In the proposed algorithm, we use the *Intermediate* approach for priority assignment. In the *Intermediate* approach, in a node, if the number of local processes is greater than or equal to the number of remote processes, then local processes will be given higher priority than remote processes, else remote processes will be given higher priority.

B. Migration-Limiting Policy :- Controlled

Migration-limiting policy decides about the total number of times a process is allowed to migrate from one node to another. In the proposed algorithm, we use the *Controlled* approach. In the *Controlled* approach, what we do, is, we associate a counter, known as *mcount* (migration count) to each process after its generation, to fix a limit on the number of times that process may migrate. Since, process migration is an expensive operation, it is not allowed to a process to migrate too frequently. To handle it, we propose that, whenever a process is created and after its priority has been assigned, the *mcount* is also initialized. If a process has a high priority, then its *mcount* will be set to 1. Since, it is a high priority process, it is not desirable to migrate it frequently. So, we allow it to migrate it only once (if necessary at all). Again, if a process has been assigned a low priority, then its *mcount* is initialized to 2, i.e. it is allowed to migrate the process only twice, and not more.

C. Load Estimation Policy:- CPU Utilization

Before we migrate a process from one node to another, we have to measure, the workload of the node from where the process will be migrated. Or, it may be the case, the local coordinator, may need to know the current workload of any of its associated nodes. In the proposed algorithm, we prefer to measure the load of a node, in terms of *CPU utilization*. The number of CPU cycles actually executed per unit of time, is known as *CPU utilization*. *CPU utilization* can be measured by setting up a timer to periodically observe the CPU state (idle/busy) [1].

D. Process Transfer Policy:- Triple-Threshold

To transfer a process form one node to another, it is necessary to decide whether a node is lightly loaded or heavily loaded. In the proposed algorithm, we use the Triple-threshold (high- intermediate– low) policy.

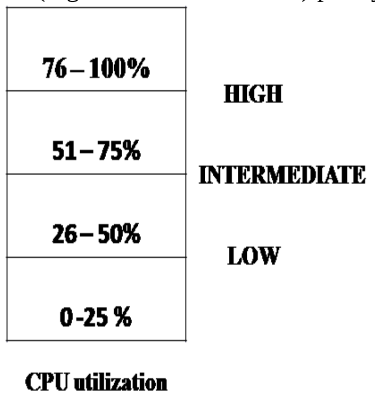

Figure 1: Different levels of CPU utilization

A decision to transfer a local process or to accept remote process is based on the following:-

i. When the load of a node is below the low mark (region 0), then new local processes (both of high & low priority) will be executed locally and for load sharing purpose, requests to accept remote processes (both high and low priority processes) are accepted.
ii. When the load of a node is above the low mark but below the intermediate mark (region 1), then new local processes (both high & low priority) will be executed locally and requests to accept remote processes (only low priority processes) are accepted.
iii. When the load of a node is above the intermediate mark but below the high mark (region 2), then new local processes (low priority processes) are executed locally and a request is sent to its local coordinator to migrate new local processes of high priority. Requests to accept new remote processes will be rejected.
iv. When the load of a node is above the high mark (region 3), then a request is sent to its local coordinator to migrate new local processes (both high and low priority). Requests to accept new remote processes will be rejected.

E. State information exchange policy :- Polling

In the proposed algorithm, we are considering dynamic load distribution policy. In dynamic load distribution policy it is implicitly required to exchange of state information among the nodes as required. There are different policies for exchange of state information – Periodic broadcast, Broadcast when state changes, On-demand exchange and Exchange by polling. In the proposed algorithm, we use the policy of *Polling [2]*. *Polling* is a better policy than the others, because, all other methods use broadcasting technique. Consequently, if the number of nodes is large in a distributed system, then broadcasting messages throughout the entire system would result into a massive traffic and network congestion. So, it is better to use the polling method. According to this policy, there is no need to exchange of state information between nodes, without necessity. Rather, when a node needs the cooperation of some other nodes, it can search, by randomly polling the other nodes one by one. The polling process stops, either when a suitable node is found or a predefined poll limit L_p is reached. In the latter approach, the conventional method is to set the poll limit (L_p) to a positive integer, and decrease the count after each unsuccessful attempt. If the poll limit is reached to zero, then the process will be executed at that node, where it was generated. Now, in this conventional approach, a little question may arise, that,

a. What will be Value of Poll Limit (L_p)?

In the proposed algorithm, we have suggested to set the poll limit to the number of local coordinators in the system.

F. Location Policy:- Threshold

Location policy determines the destination node of the process, which is decided to migrate from its current location. In the proposed algorithm, we use the method of threshold in this respect. Whenever a decision is made to migrate a process from its current location, a message is sent to its local coordinator. If the local coordinator is unable to fulfill the request within its cluster, then it sends message to other local coordinators, one-by-one. In the proposed algorithm, we implement the location policy as follows:-

Case 1:- If a node is in region 3 [(76-100%), as mentioned above], then it would send request to its local coordinator, informing that it is in region3 and wants to transfer processe(es) (high priority processes with mcount=1 and low priority processes with mcount=2 or mcount=1; as decided by the process transfer policy). Then, it is its local coordinator's duty to find a suitable node which is in region0 or in region1, and transfer some processes.

Case 2:- If a node is in region 2, then it would send request to its local coordinator, informing that it is in region2 and wants to transfer processes (as decided by the process transfer policy; high priority processes with mcount=1;) Then it is its local coordinator's duty to find a suitable node which is region0.

Case 3:- If a node is in region 1, then it would not send any request to its local coordinator.

Case 4:- If a node is in region 0, then it would not send any request to its local coordinator.

As mentioned above, if the local coordinator is unable to satisfy request of its own cluster, then it sends request one-by one to other local coordinators, in the system, until the pending requests gets satisfied.

The algorithm can best be explained by an example.

Consider a system having 20 nodes. According to the proposed algorithm, we follow the semi-distributed structure of the nodes. So, here *n*=20, i.e. total number of nodes.

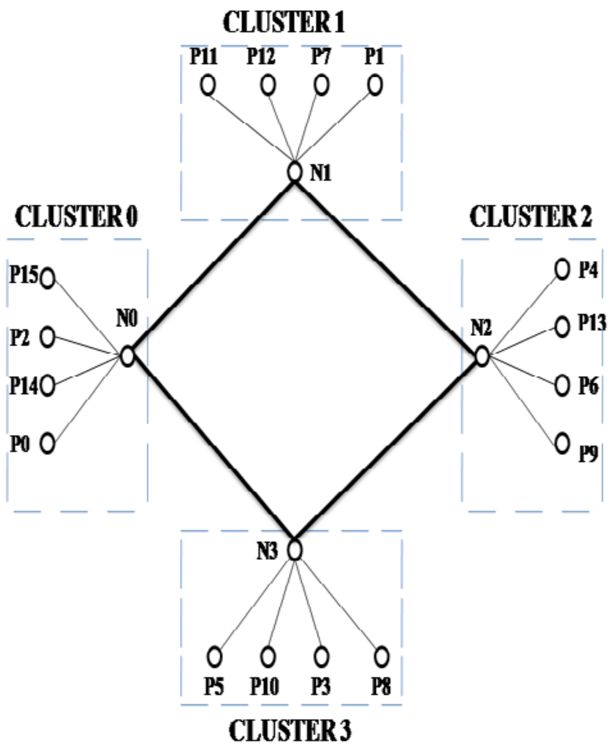

Figure 2: Semi-Distributed structure of a Distributed System

As seen in the above diagram, the nodes are logically organized in a semi-distributed structure. There are 20 nodes. So, according to the proposed algorithm, the number of local coordinators will be $floor(\sqrt{20}) = 4$, and each coordinator takes care of at the most $floor[20 \div floor(\sqrt{20})] - 1 = 4$ nodes. In this diagram, N0, N1, N2, N3 are four local coordinators. Local coordinator N0 takes care of nodes- P15, P2, P14, and P0. N0, P15, P2, P14, P0 altogether form a cluster. So, in each cluster there is only one local coordinator and its associated nodes. If any of the nodes (P15, P2, P14, P0), needs service regarding process migration, load balancing and load sharing, it only consults to their local coordinator (N0). Also, if N0 needs to fulfill requests from any of these nodes (P15, P2, P14, P0), it then, first search for the solution within its own cluster. If the request could not be satisfied, it (N0) then consults other local coordinators (N1, N2, and N3) one-by-one, until it gets satisfied. If there is no suitable node for process migration, then the process will be executed where it generated.

In the proposed algorithm, we have suggested, that, in a cluster, each node (normal nodes) maintains two tables.

a. *Local Table*: -

A local table contains information about the nodes (including coordinator) and their status. For example in cluster0, each node (P15, P2, P14, and P0) contains the following local table.

Table 1: Local table

Nodes	Status
N0	Coordinator
P15	Alive
P2	Alive
P14	Alive
P0	Alive

b. *Global Table* :-

A global table contains information about all the local coordinators present in the system. For example, in cluster0, each node (P15, P2, P14, and P0) contains the following global table.

Table 2: Global Table

Local Coordinators
N0
N1
N2
N3

On the other side, each local coordinator (N0, N1, N2, N3) also maintains the same global table. The purpose of keeping this table will be discussed soon. The working of the algorithm is as follows:- Consider a scenario, process P0, in cluster0 becomes heavily loaded, i.e. it is in region 3. P0 then sends a message to its local coordinator, informing that it became heavily loaded (i.e. it is in region 3) and try to get rid of some processes (having high priority with *mcount* =1 and processes having low priority with *mcount*=2 or *mcount*=1). After receiving this message from P0, local coordinator (N0) asks other nodes in its cluster, i.e. nodes P15, P2, and P14 to send their state information. According to the proposed algorithm, local coordinator N0 is searching for a node which is currently in the region0 or in region1. After receiving all the state information messages form rest of the nodes, P0 will decide which node is suitable for migration (i.e. which node is in region0 or in region1). If the desired node is found within the cluster, then the search stops. But, if local coordinator could not be able to find a suitable node within the cluster, it then sends a request message to other local coordinators (one-by-one); informing about its necessity. For instance, in this case, N0 sends request to the local coordinator N3, and also informing that it is in search for a node which is in region0 or in region1.
Now, local coordinator N3 would asks it local nodes for their corresponding state information. If a desired node is found, N3 locked that node, so that it cannot take any other remote processes from other node, for that moment.

N3 informs N0, and eventually some high priority and low priority processes (as mentioned previously) are migrated from P0 to the destination node. After the migrated processes complete their execution at the remote node, the lock has been released. Again, if any node, say, P9 (in cluster 2) is in region2, it then sends a request message to its local coordinator and proceeds similarly as described just above.

IV. DISCUSSION

In the discussion of the above mentioned algorithm, I would like to mention few points-

A. The semi-distributed approach that was mentioned above is beneficial in the manner; it removes the disadvantages that are in pure centralized approach and also in pure distributed approach.

In the pure centralized approach, the responsibility of dynamic scheduling physically resides on a single node (known as central coordinator). As a result, all requests for process scheduling are handled by the central coordinator. The problem with this approach –

a. If the centralized coordinator fails, the entire system would collapse [1].
b. Another problem with this approach, is the bottleneck at the central coordinator, since, each and every node sends any request (regarding scheduling) to the central coordinator. As a result, traffic congestion will occur in the communication network.
c. The third problem with the centralized approach is that, the overload of the central coordinator. If the number of number of nodes increase, then it would be tough for central coordinator to maintain such large number of nodes.

In case of pure distributed approach, the work involved in making process assignment decisions is physically distributed among the various nodes of the system. The problem with this approach –

a. Since, the system is pure distributed in nature, it may not have quick decision-making capability, which is an extremely important aspect of a good global scheduling algorithm.
b. Since, the system is purely distributed in nature, then to take any scheduling decision; it becomes necessary to collect state information of all the nodes in the system. That may result into following sub problems-

1. Collecting state information of all other nodes of the entire system is a time consuming task; delay may also occur.
2. Collecting state information of all other nodes of the entire system is also communication overhead.
3. In a distributed environment, information regarding the state of the entire system is collected typically at higher cost, than in a centralized system. The overhead is increased in an attempt to obtain more information regarding global state of the system; the usefulness of that information is decreased due to both the aging of the information being gathered and the low scheduling frequency as a result of gathering and processing that information.

To overcome these above mentioned problems, I suggested a semi-distributed structure, as mentioned above. It is neither pure centralized approach, nor purely distributed. If we have close look at the structure, then it can be seen, a scheduling decision can be taken in two ways – 1) locally and 2) globally. That is, a local coordinator can take a scheduling decision on the basis of information within its cluster (i.e. locally), or it can do it globally, i.e. referring to other local coordinators. Hence, the proposed algorithm removes the complexity of overhead of gathering state information across the entire system (i.e. in pure distributed environment). As we can clearly see in the proposed algorithm, when we do require to gather state information of other nodes, a local coordinator does not ask each and every node in the entire system, Rather, it just asks other local coordinators (one-by-one), and it up to that local coordinator to collect the state information of its cluster's. Again, the requested local coordinator is not going to search for other nodes, only interested in its own cluster. Hence, the proposed algorithm provides an optimal system performance with minimum communication overhead and also minimum global state information gathering overhead.

B. The proposed algorithm also ensures fairness of service. Consider a scenario, where a single node (which is in region 0), is selected for process migration by two (or more) different local coordinators. In that case a collision may occur. We solve this problem by locking a node (which is selected as destination node for process migration), and that lock can explicitly do by any one local coordinator at a time. Once the remote process (es) (processes which are migrated & executed at remote site) complete their execution, the lock has been released.
C. The above mentioned algorithm ensures stability, in the sense; it would not result into ***processor thrashing [2].***
D. The proposed algorithm also supports scalability. Nodes can easily be added into the system and accordingly local coordinators will be elected, along with its associated nodes.
E. The proposed algorithm has a better fault tolerance capability, compared to other approaches (pure centralized & pure distributed). If a node fails within a cluster, then it would not affect other clusters in the system. For instance, in the above diagram, if node P7, somehow goes down, then it would only affect its home cluster (i.e. cluster 1), and does not affect any other clusters too. So, we can see that the effect of failure is confined only within a certain portion of the system, does not spread away and also the effect is nominal with respect to the entire system performance. Again, if any of the local coordinators goes down, it does not affect the entire system significantly. Suppose, if the local coordinator N2 (in cluster 2) goes down, then the local node (within cluster2), who noticed it first, becomes the new coordinator of this cluster (cluster2). For instance, node P4 is the first one, noticed that N2 goes down, and becomes the new local coordinator. It then sends a message to other nodes (P13, P6, and P9) within its cluster, informing that P4 is the new local coordinator, with the help from its own local table; since from local table P4 gets to know who the other members in the cluster are. After its local announcement, P4 performs a global announcement, by updating its global table (making an entry, P4 as the new local coordinator) and send a copy of this updated table to other local coordinators (N0, N1 and N3), and in turn as acknowledgments are sent to P4.

V. CONCLUSION

From the above discussion, it is clearly seen; the proposed algorithm ensures the desirable features of a good global scheduling algorithm. That is, it supports –

a. Dynamic nature of the algorithm.
b. Quick decision-making capability.
c. Optimal system performance with minimum of global state information gathering.
d. Stability
e. Scalability
f. Good fault tolerance capability
g. Fairness in service.

VI. REFERENCES

[1] Tanenbaum A.S., Distributed Operating System, Pearson Education, Third Edition,2007.

[2] Sinha P.K., Distributed Operating Systems Concepts and Design, Prentice-Hall of India Private Limited, 2008.

[3] Attiya H., and Welch J.,, Distributed Computing Fundamentals, Simulation and Advanced Topics, John Wiley & Sons, Inc Publication, Second Edition, 2004.

[4] Kshemkalyani A.D., and Singhal M., Distributed Computing principles, Algorithms, and Systems, Cambridge University Press, First Edition, 2008.

[5] Daniel Grosu, Anthony T. Chronopoulos "Algorithmic Mechanism Design for Load Balancing in Distributed systems", IEEE TANSACTIONS ON SYSTEMS, MAN, CYBERNETICS, VOL. 34, NO. 1, pg: 77-84, FEBRUARY 2004.

[6] Hwa-Chun Lin, and Raghavendra "A Dynamic Load Balancing Policy With a Central Job Dispatcher (LBC)" ,pg:148-158, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. 18 No. 2, February 1992.

[7] Niranjan G. Shivaratri , Phillip Krueger , and Mukesh Singhal "Load Distributing for Locally Distributed Systems" , Ohio State University, IEEE, pg 33-44, December 1992.

[8] M.D. Feng , C.K.Yuen "Dynamic Load balancing on a Distributed System" ,pg:318-325, IEEE 1994.

[9] Gupta D., Bepari P., "Load sharing in Distributed Systems", Dept. Computer science & Engineering, IIT Kanpur pg: 1-16.

[10] Yung-Terng Wang, Robert J. T. Morris "Load Sharing in Distributed Systems",IEEE TRANSACTION ON COMPUTERS, Vol c-34, No. 3, pg: 204-217, March 1985.

[11] Smith J.M., "A Survey of Process Migration Mechanisms",pg: 1-13, Columbia University, 22May,2001.

© 2010, IJARCS All Rights Reserved