2009

Deploying CPU load balancing in the Linux cluster using non-repetitive CPU selection

M. Shoaib Jameel
Sikkim Manipal Institute of Technology

Muruganant Marimuthu
University of Wollongong, murugana@uow.edu.au

Tejbanta Chingtham
Sikkim Manipal Institute of Technology

International Journal of Computer and Electrical Engineering, Vol. 1, No. 2, June 2009
1793-8163

Deploying CPU Load Balancing in the Linux Cluster Using Non-Repetitive CPU Selection

M. Shoaib Jameel, M. Murugananth, and Tejbanta Singh Chingtham

Abstract— **Maintaining load balancing in a computing cluster is an evident problem in distributed systems and research in this field is not new. The challenges in designing the load balancing algorithms are immense. This paper lists some of those challenges in the design of CPU load balancing algorithm and provides solutions to some of them. The algorithm considers one node in the cluster as the Master Server and another as the Load Balancer. The master server maintains the CPU and IP information of each machine. The nodes in the cluster send their CPU status and IP information to the master server after every 30 seconds. The implementation solves "readers-writers" problem exclusively using sockets. If a number of requests are sent before the next central database update, the load balancer selects other less busy nodes in the cluster. This ensures that all nodes are allocated with the new tasks coming from remote systems, thereby maintaining a load balance among the CPUs. This implementation is highly fault tolerant and reliable, guaranteeing a high probability of task completion. Results show that this scheme handles task allocation in much optimized way and with fewer overheads. The implementation can handle CPUs ranging in numbers from 1 to 255.**

Index Terms—**CPU Load Balancing, Non-repetitive CPU Selection, Linux Cluster, Readers-Writers Problem, Fault Tolerance.**

I. INTRODUCTION

Computing clusters [57] are highly preferred these days owing to the fact that they can be cheaply setup using the desktop PCs, open source and free software. This is the reason why several research institutes and academic centers prefer computing clusters rather than purchasing supercomputers. Cluster computing provides more reliability [6], [7] and [8], availability [40], fault tolerance [1], [3], [4] and [5] and replication [39]. It does not hurt the task completion if some of the CPUs in the cluster are down. CPU load balancing [2] deals with selecting that CPU in the cluster that is minimally loaded at a given time i.e. the time when a task arrives for processing. CPU load balancing [9] and [10] can be implemented as static scheme [2] or dynamic balancing system [2]. Designing static load balancing is much easier and faster to implement. Dynamic load balancing [11] system is far more complex, CPU and network resource consuming.

The challenges in deigning the CPU load balancing scheme for a computing cluster are immense. One has to deal with the problems of process migration, maintaining lesser overhead of the CPU balancing algorithms itself, the file synchronization condition, monitoring which CPU in the cluster is down, addition of new CPU to the cluster, heterogeneity of the operating systems in the cluster and plenty more.

The scheme described in this paper maintains a prior knowledge of the number of CPU's in the cluster along with their usage information. The task becomes more complicated when it comes to designing algorithms for a heterogeneous computing cluster, but here we deal with only homogeneous environment consisting computers loaded with GNU/Linux operating system. This scheme is designed from scratch in C programming language owing to efficiency reasons.

The scheme described in this paper conforms to the dynamic load balancing setup. The most unique aspect of this implementation is the solution that it provides to the readers-writers problem using sockets. It shows that apart from file locking, sockets can be used effectively to solve the problem of file read/write in a shared memory environment. This implementation is highly fault tolerant and guarantees task completion.

In this research, the following questions are addressed:

- How can the Readers-Writers problem be solved in scenarios where file locking mechanism is not feasible?
- Can better performance and throughput be achieved without the use of specialized packages for setting up clusters like MOSIX etc?
- How can the reliability and availability of the cluster be increased?
- How can we design optimized algorithms that use fewer overheads for CPU allocation and task migration and more on doing priority computations?
- How can network consumption of the balancing algorithms be reduced?

Manuscript received September 13, 2008.

M. Shoaib Jameel was with the Department of Computer Science and Engineering, Sikkim Manipal Institute of Technology, Majitar, Rangpo, East Sikkim - 737132 INDIA. He is now with the Department of Research and Development/Scientific Services, Tata Steel Limited, Jamshedpur India (corresponding author, phone: +919234502858).

M. Murugananth is with the Department of Research and Development/Scientific Services, Tata Steel limited, Jamshedpur, India. Phone: +919934302578

Tejbanta Singh Chingtham is with the Department of Computer Science and Engineering, Sikkim Manipal Institute of Technology, Majitar, Rangpo, East Sikkim - 737132 INDIA. (phone +919734916135).

II. RELATED WORK

Research related to enhancing the processing capability of a computing setup has been discussed before [12]. The major challenge in setting a cluster-based system is the design and development of algorithms that maximizes performance using an optimality criterion [13] and also minimizes the total execution time [28], [29] and [30]. The issue of load distribution emerged when distributed computing systems and multiprocessing systems began to gain popularity. Over the years, several algorithms related to the problems in load balancing in computing clusters have been proposed [14], [15], [31], [32] and [33]. In [31] and [32], it is assumed that the processors are always lightly loaded, but normally the load varies in an unpredictable manner in a workstation. In [33], task migration has been discussed but it involves a large amount of communication overhead. Numerous strategies for static and dynamic load balancing have been developed, including recursive bisection (RB) methods [16], [17] and [18], space filling curve (SFC) partitioning [19], [20], [21] and [22], graph partitioning [17] and [23] and diffusive methods [24], [25] and [26].

Load balancing scheme has also been applied on the web servers [34], [35], [36], [37] and [38] for efficient user request handling. A distributed web server system is any architecture consisting of multiple web-server hosts, distributed on LAN and WANs with a mechanism to spread incoming client requests among the servers.

The work described here discusses an implementation built from scratch in C programming language owing to the efficiency reasons. The most important accomplishment of the design has been the innovative way in which the readers-writers problem has been solved and the allotment of CPUs (nodes) using non-repetitive selection procedure. The design does not make use of any third party applications for setting up distributed systems like monitoring tools e.g. Ganglia [27]. The design itself monitors the CPUs that are up and running and informs the administrator in case any of the CPU(s) is/are down. This design can support CPUs ranging from 1 to 255 but the algorithms have been tested with CPUs five in number. Though the architecture in this balancing scheme has been developed keeping in mind the heterogeneity of the systems that connect to the master server for task completion but the same design can be implemented on any cluster computing setup. This work differs from the other works in the way this design solves readers-writes problem and non-repetitive CPU selection.

III. LINUX CLUSTER SETUP

The cluster consisted of **5** nodes installed with GNU/Linux the operating system. The Linux kernel was recompiled with minimum device driver support. Features like multimedia support, wireless networking and the like were not included in the final compiled kernel. The requirement was to have a Linux kernel, which consumed least CPU and memory in other jobs.

This cluster did not make use of any existing tools or software that support cluster computing. The load balancing algorithms were designed from scratch after studying the underlying architecture of this computing cluster and the set of tasks for which it would be used for. Some examples of the tools for cluster computing are MOSIX [41] and [42]. Mosix is a set of adaptive resource sharing algorithms that are geared for performance scalability in a computing cluster of any size, where the only shared component is the network. The core of the Mosix technology is the capability of multiple nodes (workstations and servers, including SMP's) to work cooperatively as if part of a single system. Software packages for process allocation include PVM [43] and LSI [44].

LSF [45] and Extreme Linux [46] provide related services. These software packages provide an execution environment that requires an adaptation of the application and the user's awareness. They include tools for initial assignment of processes to nodes, which sometimes use load considerations, while ignoring the availability of other resources, e.g. free memory and I/O overheads. These packages run at the user level, just like ordinary applications, thus are incapable to respond to fluctuations of the load or other resources, or to redistribute the workload adaptively.

The computing cluster described here was used for executing modeling computations that were highly CPU intensive jobs. Due to heterogeneity of the remote systems that connect to the cluster for computations, an adapter was designed and coded. The entire process is shown in **Fig. 1.**

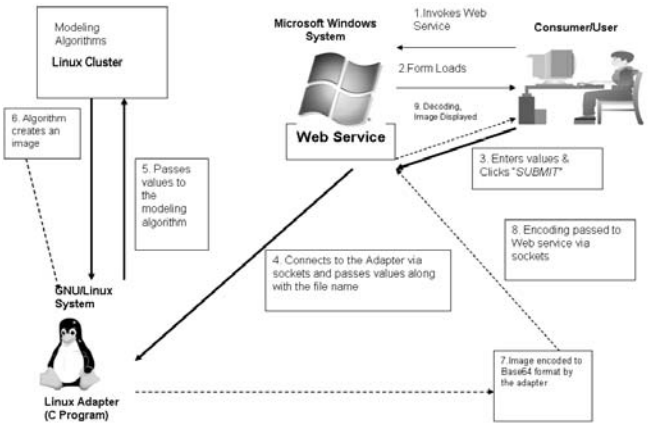

Fig 1: A detailed flow of the entire system and the role of the Linux cluster in the operation. The entire communication was carried out using BSD socket API.

Web services were first created for different modeling algorithms. The web services had forms, which a user filled with some real numbers. The adapter was written in C and acted as a bridge between the web services hosted on the Microsoft Windows based system and one of the nodes in the computing cluster. The Linux adapter listened to a port for incoming connections from the web service. The real values from the form were concatenated and each value being separated by a special character, for example 0.0001#0.343. This concatenated input stream was then passed to the adapter installed on a Linux machine via sockets. The adapter subsequently parsed the entire value stream and segregated the real number values. The segregated values were subsequently fed to the modeling algorithm for processing in the computing cluster. It has to be mentioned that the web service along with the real numbered values also passed the name of file to be created by the processing node in the cluster and an identifier value of the modeling algorithm,

which was to be invoked at the computing cluster. Invocation of the modeling algorithms was done internally by modeling invocation system. Two adapters were designed, one which worked externally and communicated directly with the web service and other adapter that was internal to the computing cluster. The internal adapter was similar to the external one except that it had the modeling invocation module.

IV. Assumptions

1. Load Balancer server was always up and running. If the load balancer server went down the entire architecture failed.
2. It did not matter if any machine in the cluster other than the load balancer server crashed.
3. If all the machines in the cluster were down except the load balancer, the load balancer server took over the modeling calculations.
4. The internal Linux Adapter needed to be up and running on all the machines (nodes) in the cluster and all adapters must use a common port.
5. The client programs (programs that send IP and CPU status to the master server) should be installed on every machine in the cluster except the master server.
6. The client programs must use one port and their connecting IP must be the IP of the master server.
7. The master server was not some special computer but it was one among the computers in the computing cluster that handled an extra task of maintaining the central database.
8. There was another program installed on another node of the computing cluster called the tunnel program or the Load Balancing Program. This program received the values from the web service and the database values from the master server.
9. The Load Balancing Algorithm decided which CPU in the computing cluster had the least load.
10. This was not a general-purpose load-balancing algorithm. This entire architecture was catered to the requirements of the modeling computations and was hooked to the Linux Adapter.
11. This architecture would run on any Linux, UNIX and Solaris based systems.

V. THE MASTER SERVER

One node in the cluster was chosen as the master server. The job of the master server was to maintain a database, in a text file, called the Central Database. The central database contained the CPU status information and IP addresses of all the machines in the computing cluster.

The master server continuously listened to a port for any incoming connections. Through this port, all the machines (nodes) in the cluster connected and sent their CPU status and IP addresses after every 30 seconds. The master server also had another database where the administrator fed in the IP addresses of the machines in the cluster manually. Therefore, whenever a new machine was added to the cluster this database needed a manual update. After every central database update, the master server checked as to which node in the cluster was up and sending its status information. This check was made by comparing the IP addresses in the central database, which was automatically updated after every 30 seconds, with that of the manual database that was maintained by the administrator. In order to optimize the above scheme, the master server first made a check of the number of IP addresses that were there in the central database. If there was a difference in the number of IP addresses between the newly updated central database and the manual database, the master server's IP address searching module searched for the IP, which had not sent the values. Subsequently, the master server checked two times for that node in succession i.e. two times when the new values had been received after 30 seconds. If both the times, that node had not sent the CPU status to the master server, the master server immediately sent an urgent e-mail for attention to the administrator. This design ensured that whenever a node in the cluster was down, the administrator came to know about it immediately and action could be taken as soon as possible.

Table 1: The structure of the Central Database consisting of the CPU status and IP address of the machines. This database was updated after every 30 seconds. In order to solve the readers-writers problem the database was read into an array and passed onto the load balancer, which stored the contents in memory until next update comes.

IP	CPU
120.44.83.21	89.7
10.9.8.7	10.43
1.2.3.4	0.004
5.4.3.2	1.30
133.0.0.43	90.08

It was noticed during the implementation phase that if the administrator did not make a note of the CPU which was down for several hours, the master server used to send the same mails after every two updates. In order to solve the above problem it was decided to mail the administrator after every 3 hours, if the same IP was noticed to be down after the two central database updates. A high-level architectural diagram of the master server is shown in the Fig 2.

VI. The Load Balancer

The load balancer did CPU allocation procedure. After request for computation had been received by the load balancer, it sent the input stream to the least busy CPU in the cluster first. It also applied the non-repetitive CPU selection algorithm.

The load balancer continuously listened to a port. Two different applications connected to the load balancer through this port, at different times. One was the external adapter that sent the input stream, to be processed by the modeling algorithms and other was the master server that sent the entire central database in a very large array. CPU and IP information in the central database was stored in a single line for each node. The master server read the central database line by line and added a special character after every line. This processed stream (by the master server) was then passed to the load balancer. The stream was stored in the load balancer in a very large array, which was dynamically allocated at runtime.

The design of the load balancer is given in the Fig 4:

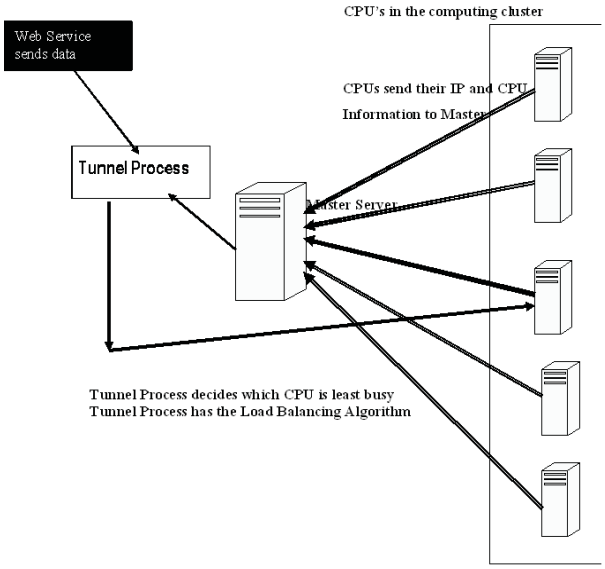

Fig 2: Diagram showing the complete detailed architecture of the cluster setup.

VII. SOLVING READERS-WRITERS PROBLEM

Mutual exclusion is a sufficient mechanism for most forms of synchronization, but it introduces serialization that is not always necessary. Readers-Writer synchronization [47] relaxes the constraints of mutual exclusion to permit simultaneously, so long as none of them modifies the file. Operations are separated into two classes: writes, which require exclusive access while modifying the data structure, and reads, which can be concurrent with one another (though not with the writes) because they make no observable changes.

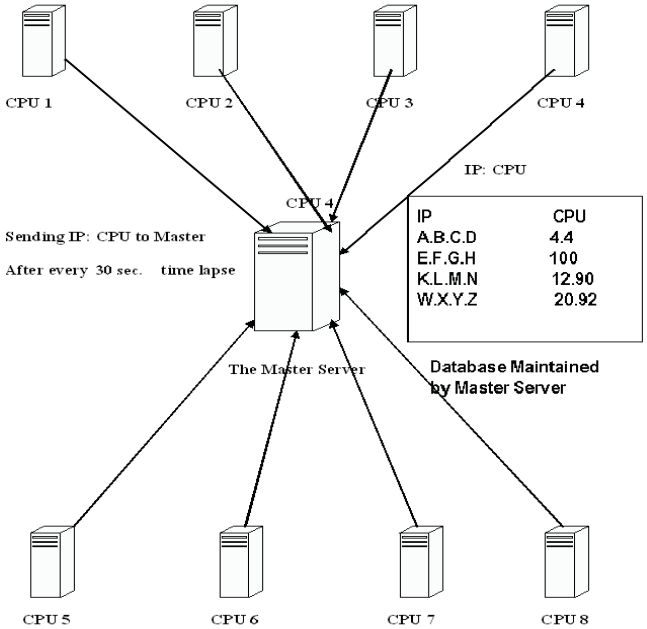

Fig 3: Figure depicting the High Level diagram of the master server. The CPUs in the cluster send their IP and CPU status information after every 30 seconds. The 30 seconds time frame can either be increased or decreased depending upon administrator's choice and need.

Works undertaken previously consider using mutual exclusion [53], [54], [55] and [56] or file locking mechanisms [48], [49], [50] and [51] for synchronization. But the work descried here applied sockets to solve the problem of readers-writers.

During the developmental stages, it was noticed that as the master server was on the course of updating the central database, the load balancer simultaneously read the database. This crashed the load balancer as the data in the central database was inconsistent.

File locking was not an option owing to the fact that if the number of CPUs in the cluster was increased, then most of the time the central database (the text file) would be locked as the update process would be under way. Hence, load balancer could not read it for a long time.

The problem of readers-writers was resolved in an innovative way, where we built our own version of mutual exclusion semantics. The following points are worth to be noted in the design of this scheme:

- listen() function (used in socket programming in C and some other computer programming languages) allows only one client to connect at one common port at one given time. If there were a number of requests from different clients, listen() function queues the new requests in the wait queue [52].
- Two different dynamic arrays could be implemented in the load balancer One array can store the central database values and the other array can store the input stream received from the internal adapter. This would eliminate the file read/write.
- If the input stream was received from the web service, then the load balancer checked for the least busy CPU in the cluster and the corresponding IP, which was stored in another dynamic array in the load balancer. The new task was then migrated to the least busy CPU in the cluster by the load balancer.
- If the load balancer received socket connection from the master server, the array containing the central database values in the load balancer was updated.

It was observed that as the central database was updated, a point was reached when it was complete and consistent. This was the point when all the nodes in the cluster had replied to the master server with their CPU and IP information. After the central database became consistent, the database was read and passed to the load balancer through socket. The format that was passed to the load balancer was *IP1 CPU1%IP2 CPU2%IP3 CPU3…n*, so that the parsing algorithm can know the start and end of the IP and CPU status information in this input stream.

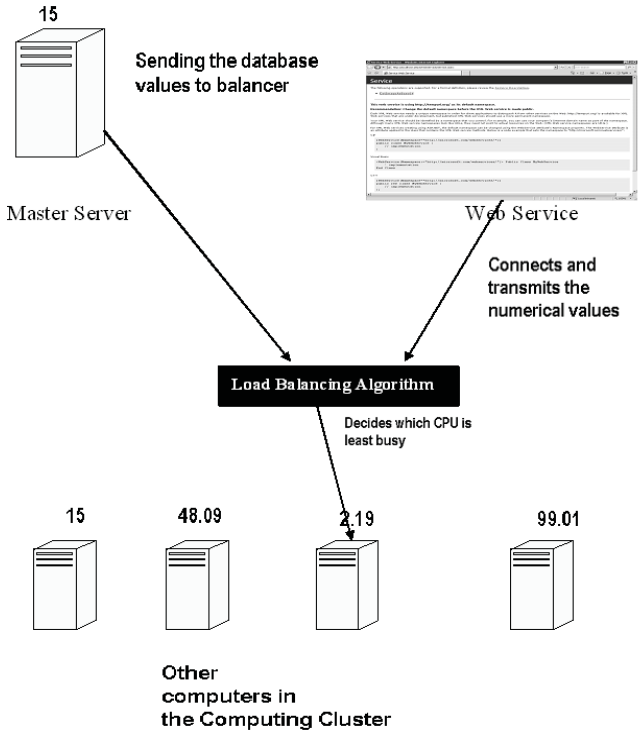

Fig 4: The figure depicting the load balancer in the Linux cluster. The numbers at the top of the computers represent their current CPU usage in percentage. Two different applications connected to the load balancer at different times. The design ensured that if both the applications try to connect to the load balancer simultaneously, one is put on wait and the other is given an opportunity to execute in the code section of the load balancer. The load balancer kept the central database contents in memory.

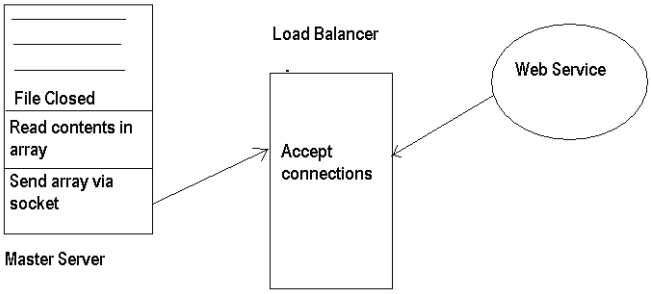

Fig 5: An illustration showing an implementation that solved the readers-writers problem.

The algorithm for the readers-writers solution is as follows:
Step 1: OPEN the central database file in write mode.
Step 2: Wait for 30 seconds until the computers in the cluster respond with their CPU and IP values.
Step 3: CLOSE the central database file.
Step 4: OPEN the central database in READ mode.
Step 5: READ the contents of the central database and append special characters after each line.
Step 5: STORE the central database contents in a string array.
Step 6: CONNECT to the load balancer and send the entire array via socket.
Step 7: CLOSE the central database file.
Step 8: REPEAT Step 1

Two different applications connected to the load balancer

- The web service.
- The master server.

When the applications connected at different times, the following was done:

1. Web Service: When the web service connected, the load balancer received the concatenated real values separated by some special character. This input stream was stored in an array. Subsequently, the algorithm searched another array consisting of the contents of the central database, which the master server had sent it previously, for the CPU that was least busy in the entire cluster. The corresponding IP was parsed and the load balancer forwarded the input stream to the internal adapter corresponding to that IP for modeling calculations.
2. Master Server: When the master server completed the operation of central database update, it connected to the load balancer on the same port as the web service using sockets and transferred the entire database. The balancer stored this data in dynamically allocated memory chunk, until the master server connected again and sent the updated values. This array update occurred after every 30 seconds in the load balancer.

VIII. NON-REPETITIVE CPU SELECTION ALGORITHM

Problem Definition: It was noticed that when several users used modeling computations simultaneously at one given time repetitively, only one CPU in the cluster used to get the all the input stream until new updated values from the master server was received by the load balancer. The waiting task queue for one node used to get very long as compared to the rest of the nodes. This is because the load balancer does the task allocation based on the notion that only that CPU would get the input stream from the web service, which had the lowest CPU usage in the central database. This resulted in overloading the processor (node) of the least busy node in the cluster, while others were relatively less busy.

One solution to this would be to reduce the central database update time in the load balancer from 30 seconds to 5 seconds. But, this would consume network and CPU resources, which was not feasible as we had to dedicate the computation to the modeling algorithms, which was the top priority.

In order to resolve the problem, the load balancer had a scheme that applied the algorithm of non-repetitive CPU selection. This meant, choosing different processors on each new incoming request, until the next update in the central database occurred. This algorithm worked as follows:

There were four different modeling algorithms. Each of the algorithm's runtime CPU consumption was noted. At the end, an average value was taken which depicted as to how much CPU processing each of the modeling algorithm took. This process was done using the Linux's *top* command. The following average values were finally used:

Let modeling algorithm 1 is written as M1. Similar goes for other modeling algorithms i.e. M2, M3 and M4. The average CPU consumption of each of the modeling algorithm was as follows:

M1 = 30%, M2 = 50%, M3 = 10%, M4 = 20%

When the request was made by the user after central database array update in the load balancer, the least busy CPU in the cluster was allotted the task of computation. If other user requested for computations and the same CPU

values in the load balancer array are still there i.e. no update has come from the master server, then the following was done:

Let us suppose that the computers in the cluster are denoted by C1, C2…C8 i.e. 8 nodes and request for modeling computation was made by a user. Suppose the current central database CPU values in the load balancer array are as follows:

C1 = 10%, C2 = 12.5%, C3 = 20%, C4 = 80%, C5 = 2%, C6 = 99%, C7 =.3% and C8 = 100%

When the request for computation was made for the first time after the load balancer got the updated values from the master server, the load balancer searched for the minimum CPU usage in the database and allocated the task of computation to that CPU. In this case, C7 was allocated the task for computation.

Now, another user or same user sent another request for computation, the non-repetitive algorithm did the following calculations:

Step 1: Determine the type of modeling algorithm invoked by the user.

Step 2: Say, M2 was the algorithm invoked by the user, while M1 was already executing in C7.

Step 3: M2 took 50% of the entire CPU. Next least busy CPU in the cluster was C5 with 2% usage. Therefore, if M2 was allocated to C5, C5 usage becomes 52% (approximately).

Step 4: If M2 was allocated to C7 which itself was executing M1, the CPU usage at C7 end would have been 30 + 0.3 = 30.3% (approximately). Now, allocating M2 to C7 would mean that now the CPU computation became 30.3 + 50 = 80.3% (approximately)

Step 5: Therefore, M2 went to C5 for computation.

Step 6: Again, another user invoked M3, which took 10% of the entire CPU. C5 now was 52% (approximately) and C7 = 80.3% (approximately). Next least busy CPU was C1 with 10% usage.

Allocating task to C1 meant that 10 + 10 = 20% (approximately)

Allocating to C5 = 52 + 10 = 62% (approximately)

For C7 = 80.3 + 10 = 90.3%, hence, task migrated to C1 for computation.

Step 7: Another task came in for computation. Let us suppose this was the 7th task in succession and updated values had not yet come from the master server (30 second time frame had not completed). The algorithm checked for the next least busy CPU. The algorithm ensures that the sum of the current CPU usage and modeling computation values never exceeded 100. If this is the case, then the algorithm again began from the least busy CPU in the cluster and repeated with the next busy CPU and so on.

Consider an example, C6 = 99% and M4 = 20%. If C6 came next in the least busy CPU ordered list, this CPU was not selected for computation as 99+20 = 119.0 as 119 > 100

The algorithm again chose the least busy CPU i.e. C7 with 30.3 + 20 = 50.3% and 50.3 < 119. So, task allocated to C7. This had been done keeping in mind that by this time the first request would be about to complete or already has already completed in C7.

The following condition was considered in the current implementation.

The task completion time of the modeling algorithm: Each modeling algorithm took about 10 seconds to execute and produce the result. If the non-repetitive algorithm kept track of the task completion time of the CPUs, which had already been allocated the task of computation, then the performance would have been even better. This would mean that by the time, task one was over and another task came in for computations then task 2 could be allocated to the CPU, which had completed the computation just moments ago. Hence, the least busy CPU would be used again after first task completion.

IX. EXPERIMENTS AND RESULTS

The algorithms were written in C programming language, owing to efficiency. None of the applications discussed above are GUI based. Due to the copyright policies, the actual modeling algorithms are not shown in this paper. However, similar applications were developed, which simulated the real modeling applications. The results discussed here are screenshots of the console applications.

The experiments were performed under laboratory conditions. Some of the CPUs were overloaded with jobs deliberately to test the algorithms. The algorithms' execution time was traced and noted using the *top* command in Linux. When the load balancing algorithms were executing, we noticed their CPU consumption to be about 0.1% of the entire primary memory.

The console applications just printed the IP on the screen when the task was allocated to the CPU. The internal adapters were installed on every machine in the cluster. The values from the web services form were concatenated and passed onto the load balancer.

Below are some of the screenshots:

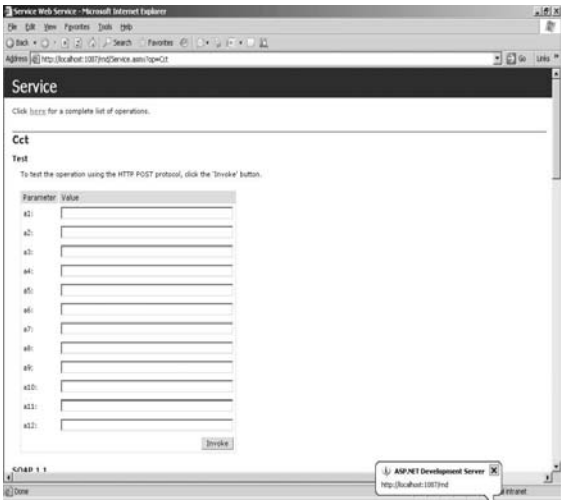

Fig 6: A screenshot showing the web service, which was a form that the user filled in with some real number values. All the values entered were concatenated in the form number#number#...#filename#algorithm_identifier.

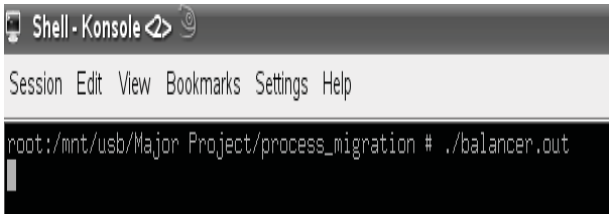

Fig 7: Screenshot of the load balancer algorithm in port listening state. As this algorithm receives request for connection it does the required execution.

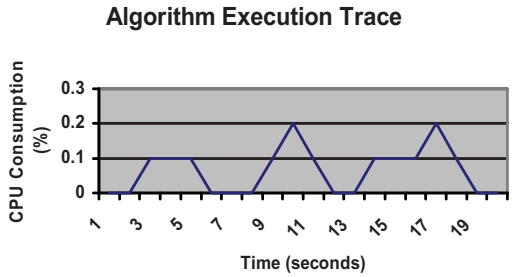

Fig 8: CPU Consumption v/s Time:

The plot shows CPU utilization of one node in the cluster, which executes the modeling algorithms. Due to non-repetitive CPU selection scheme for a very short duration the CPU utilization becomes zero, meanwhile in this state of zero processing, the CPU handles other non-modeling related jobs. This plot shows only the CPU consumption by the modeling algorithms.

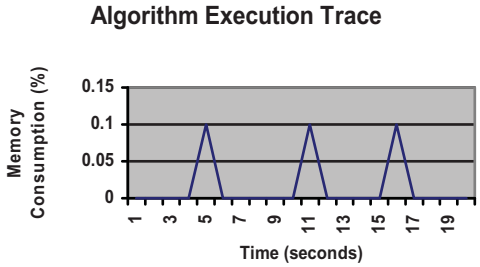

Fig 9: Memory Consumption v/s Time: Memory consumption by a node during modeling computations.

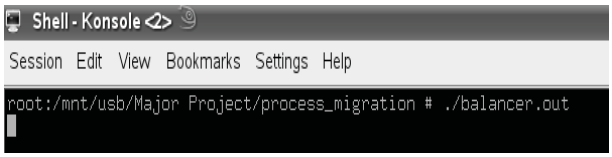

Fig 10: Screenshot of the load balancer algorithm in port listening state. As this algorithm receives request for connection it does the required execution.

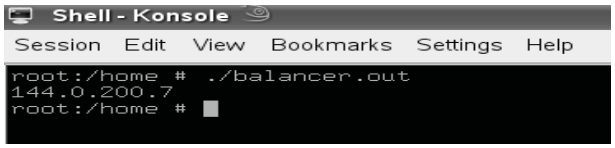

Fig 11: As connection is received from the web service, the algorithm printed the IP address of the machine, which has the least busy CPU in the entire cluster.

X. CONCLUSION

This design is highly reliable and fault tolerant. Problem occurred only when the load balancer was down, the entire system came to a halt. This problem could be resolved only when the web service is able to connect to different machines after sensing that the load balancer is down. This task has to be done at the Windows end. The best bet is to run the tunnel and the master server cluster programs on different computers in the computing cluster. Reason being, even if one of them is down the system would keep functioning without fault. The load balancer would take over the modeling calculations in such a case.

ACKNOWLEDGMENT

We would like to thank Mr. K. Krishna Raju, Information Technology Services, Tata Steel Limited, Jamshedpur, India for developing the Microsoft Windows based web services. We would also extend our sincere gratitude to Mr. Fredi B. Zarolia, Senior Manager (IT), New Initiatives, Tata Steel, Jamshedpur, India for assisting us during the course of this project. His active involvement in this research is commendable. He gave us some important insights during the implementation phase which helped us overcome some of the major hurdles. Computing and network services were provided by Tata Steel Limited, Jamshedpur, India.

REFERENCES

[1] P.K. Sinha, “Distributed Operating Systems: Concepts and Design”, Wiley-IEEE Press 1996.
[2] C. Steketee, “Process Migration and Load Balancing in Amoeba”, In the Proceedings of the Twenty Second Australian Computer Science Conference, Auckland, New Zealand, January 18--21 1999. Springer-Verlag, Singapore.
[3] B. Randell, P.A. Lee, P. C. Treleaven (June 1978). "Reliability Issues in Computing System Design", ACM Computing Surveys (CSUR) 10 (2): 123–165. doi:10.1145/356725.356729. ISSN 0360-0300.
[4] P. J. Denning (December 1976). "Fault tolerant operating systems", ACM Computing Surveys (CSUR) 8 (4): 359–389. doi:10.1145/356678.356680. ISSN 0360-0300.
[5] Theodore A. Linden (December 1976). "Operating System Structures to Support Security and Reliable Software", ACM Computing Surveys (CSUR) 8 (4): 409–445. doi:10.1145/356678.356682. ISSN 0360-0300.
[6] E. Redwine, and J.L. Holliday, Santa Clara University. [Online] Available: http://www.cse.scu.edu/~jholliday/REL-EAR.htm (reliability write up)
[7] R. Chow, and T. Johnson, “Distributed Operating Systems & Algorithms”, Addison Wesley Longman, 1997.
[8] A. Tanenbaum, “Distributed Operating Systems”, Prentice-Hall Inc, 1995.
[9] P. Werstein, H. Situ, and Z. Huang, “Load Balancing in a Cluster Computer”, Parallel and Distributed Computing, Applications and Technologies, 2006. PDCAT apos;06. Seventh International Conference on Volume, Issue, Dec. 2006 Page(s):569 – 577 Digital Object Identifier 10.1109/PDCAT.2006.77.
[10] C. Youn, and I. Chung, “An Efficient Load Balancing Algorithm for Cluster System”, Publisher Springer Berlin / Heidelberg, ISSN 0302-9743 (Print) 1611-3349 (Online), Volume 3779/2005, pp 176-179.
[11] K.D. Devine, E.G. Boman, R.T. Heaphy, B.A. Hendrickson, J.D. Teresco, J. Faik, J.E. Flaherty, and L.G. Gervasio, “New Challenges in Dynamic Load Balancing”, Reprinted by Elsevier Science.
[12] T.G. Lewis, and H. El-Rewini, 1992, “Introduction To Parallel Computer”, Prentice-Hall, Inc.
[13] C.Z. Xu and F.C.M. Lau 1997, “Load Balancing In Parallel Computers: Theory And Practise”, Kluwer Academic Press.
[14] K.M. Baumgartner, and B.W. Wah, 1991, “Computer Scheduling Algorithms: Past, Present, and Future, ” Information Science, 57-58, Pp319-345.
[15] T.L. Casavant, and J.G. Kuhl, 1994, “A Taxonomy of Scheduling in General Purpose Distributed Computing Systems.” In T.L. Casavant and M. Singhal, ed., Readings In Distributed Computing Systems, IEEE Computer Society Press.
[16] M. J. Berger, and S. H. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors", IEEE Trans. Computers C-36 (5) (1987) 570-580.
[17] H. D. Simon, "Partitioning of unstructured problems for parallel processing", In: Proc. Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications, Pergammon Press, 1991.
[18] V. E. Taylor, and B. Nour-Omid, "A study of the factorization fill-in for a parallel implementation of the finite element method, Int. J. Numer. Meth. Engng. 37 (1994) 3809-3823.
[19] M. S. Warren, and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm", In: Proc. Supercomputing '93, Portland, OR, 1993.
[20] J. R. Pilkington, and S. B. Baden, "Partitioning with space filling curves", CSE Technical Report CS94-349, Dept. Computer Science and Engineering, University of California, San Diego, CA (1994).

[21] A. Patra, and J. T. Oden, "Problem decomposition for adaptive hp finite element methods", J. Computing Systems in Engg. 6 (2).
[22] W. F. Mitchell, "Refinement tree based partitioning for adaptive grids", In: Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, 1995, pp. 587-592.
[23] A. Pothen, H. Simon, and K. Liou, "Partitioning sparse matrices with eigenvectors of graphs", SIAM J. Matrix Anal. 11 (3) (1990) 430-452.
[24] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors", J. Parallel Distrib. Comput. 7 (1989) 279-301.
[25] Y. Hu, and R. Blake, "An optimal dynamic load balancing algorithm", Tech. Report DL-P-95-011, Daresbury Laboratory, Warrington, WA4 4AD, UK (Dec. 1995).
[26] E. Leiss, and H. Reddy, "Distributed load balancing: design and performance analysis", W.M. Keck Research Computation Laboratory 5 (1989) 205-270.
[27] Ganglia Monitoring System [Online]: Available: http://ganglia.info/
[28] R. Diekmann, and B. Monien, "Load Balancing Strategies for Distributed Memory Machines", Computer Science Technical Report Series "SFB", No. tr-rsfb-97-050, Univ. Of Paderborn, Germany, p. 1-37.
[29] V. Kumar, A. Grama, A. Gupta and G. Karypis, (1994) "Introduction to Parallel Computing: Design and Analysis of Algorithms", Benjamin Cummings, New York, p. 1-597.
[30] R. Lling, B. Monien, and F. Ramme, (1991) "A study of dynamic load balancing algorithms", Proceedings of the 3rd IEEE SPDP, p. 686-689.
[31] C. K. Lee, and M. Hamdi, (1995) "Parallel image processing application on a network of workstations", Parallel Computing, 21, p. 137-160.
[32] C. K. Lee, and M. Hamdi, (1994) "Efficient parallel image processing application on a network of distributed workstations", Proc. 8th International Parallel Processing Symposium, p. 52-59.
[33] N. Nedeljkovic, and M. J. Quinn, (1993) "Data-Parallel Programming on a Network of Heterogeneous Workstations", Concurrency: Practice and Experience, 5, 4, p.257-268.
[34] L. Aversa, and A. Bestavros, "Load balancing a cluster of web servers: using distributed packetrewriting", In the Proceedings of Performance, Computing, and Communications Conference, 2000. IPCCC '00. Conference Proceeding of the IEEE International, Feb 200, p. 24-29, ISBN: 0-7803-5979-8.
[35] J. L. Wolf, and P. S. Yu, On balancing the load in a clustered web farm, ACM Transactions on Internet Technology (TOIT), v.1 n.2, p.231-261, November 2001.
[36] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic Load Balancing on Web-Server Systems", IEEE Internet Computing, v.3 n.3, p.28-39, May 1999.
[37] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic placement for clustered web applications", Proceedings of the 15th international conference on World Wide Web, May 23-26, 2006, Edinburgh, Scotland.
[38] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu, "The state of the art in locally distributed Web-server systems", ACM Computing Surveys (CSUR), v.34 n.2, p.263-311, June 2002.
[39] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu, "Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services", In Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems, pages 197–208, San Francisco, CA, Mar. 2001.
[40] Y. Saito, B. N. Bershad, and H. M. Levy, "Manageability, Availability, and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service", In Proc. of the 17th ACM Symposium on Operating Systems Principles, pages 1–15, Dec. 1999.
[41] A. Barak, S. Guday, and R.G. Wheeler, "The MOSIX Distributed Operating System, Load Balancing for UNIX", In Lecture Notes in Computer Science, Vol. 672. Springer-Verlag, 1993.
[42] A. Barak and O. La'adan, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing", Journal of Future Generation Computer Systems, 13(4-5):361–372, March 1998.
[43] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM - Parallel Virtual Machine", MIT Press, Cambridge, MA, 1994.
[44] W. Gropp, E. Lust, and A.Skjellum, "Using MPI", MIT Press, Cambridge, MA, 1994.
[45] Platform Computing Corp. LSF Suite 3.2. 1998.
[46] Red Hat. Extreme Linux. 1998.
[47] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with "readers" and "writers"", Communications of the ACM, v.14 n.10, p.667-668, Oct. 1971.
[48] Readers-Writer Lock [Online]: Available: http://en.wikipedia.org/wiki/Readers-writer_lock
[49] J. A. Trono, "A new exercise in concurrency", ACM SIGCSE Bulletin, v.26 n.3, p.8-10, Sept. 1994.
[50] C.A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: a scalable distributed file system", Proceedings of the sixteenth ACM symposium on Operating systems principles, p.224-237, October 05-08, 1997, Saint Malo, France.
[51] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux", In Proceedings of the Ottawa Linux Symposium, 2002.
[52] listen() function in C man page [Online]: Available: http://linux.die.net/man/2/listen
[53] O. S. F. Carvalho, and G. Roucairol, "On mutual exclusion in computer networks", Commun. ACM 26, 2 (Feb. 1983), 146-147.
[54] K. Raymond, "A tree-based algorithm for distributed mutual exclusion", ACM Transactions on Computer Systems (TOCS), v.7 n.1, p.61-77, Feb. 1989.
[55] V. Kumar, J. Place, and G. C. Yang, "An Efficient Algorithm for Mutual Exclusion Using Queue Migration in Computer Networks", IEEE Transactions on Knowledge and Data Engineering, v.3 n.3, p.380-384, September 1991.
[56] S. Lodha, and A. Kshemkalyani, "A Fair Distributed Mutual Exclusion Algorithm", IEEE Transactions on Parallel and Distributed Systems, v.11 n.6, p.537-549, June 2000.
[57] Cluster (Computing) [Online]: Available: http://en.wikipedia.org/wiki/Computer_cluster