Neighbour Aware Random Sampling (NARS) algorithm for load balancing in Cloud computing

Ariharan V,
Central Research Laboratory,
Bharat Electronics Limited,
Bengaluru.
ariharanv@bel.co.in

Sheeja S Manakattu,
Central Research Laboratory,
Bharat Electronics Limited,
Bengaluru.
sheejasmanakattu@bel.co.in

***Abstract*— Load balancing plays a vital role in Cloud computing to enhance throughput, optimize resource use and reduce response time. The main features to be considered while selecting a load balancing algorithm for cloud is the ability of the algorithm to address distributed network, dynamic environment and self-regulation. Biased random sampling is one such algorithm; it allocates jobs by performing a random walk in the network. The selection of neighbour is uniformly distributed among the neighbour nodes in case of biased random sampling algorithm. Improved version of random sampling algorithm uses cost based load computation to select node for random walk [1]. This paper introduces neighbour awareness and prediction mechanisms to further improve the selection process of nodes for random walk. The proposed algorithm selects the least loaded node from the neighbour list for the random walk. This can be achieved by computing probability of each neighbour based on perceived load of the respective neighbour. Thus the probability of choosing lightly loaded node can be increased and hence the job waiting time can be decreased further.**

Keywords*— *Cloud computing, load balancing, random sampling, neighbour awareness, probabilistic algorithms.

I. INTRODUCTION

Cloud computing is an emerging technology that improves the performance of web services, database management, scientific computations and computation intensive processes to perform multitude of computational demands and maintain redundant sources for availability. As the number of physical machine increases, various process management problems such as load balancing, computational integrity, atomicity of data, etc. need to be addressed. For better availability, response time, throughput and reliability, load balancers are used in the network.

A load balancer is a key component in any cloud architecture which distributes incoming service request among the available backend servers as shown in Fig.1. The load balancing algorithm for load balancers can be a generic one or specific to particular cloud category. For example, algorithm for a database load balancer and an FTP load balancer can be different. To prevent single point failure, database servers are replicated. This redundancy also helps to increase the availability of data and decrease the rate of failure if the database servers are spread across the globe. The sole purpose of a database load balancer is to distribute the incoming requests among the available database servers. Besides providing fault tolerance and availability through redundancy, an FTP service based cloud should try to reduce the file transfer time. Hence a load balancer used in an FTP server should be able to route the request such that the delay parameters of transmission from server to client is minimal to reduce the file transfer time.

A. Motivation

The increasing amount of nodes in the cloud and their distribution across the world makes it difficult to maintain a centralized load balancer. Being an entry point to the cloud, it is also better to maintain distributed network of load balancers to avoid cloud failure. Hence the load balancing algorithm should have the capability to work on self-regulated distributed environment.

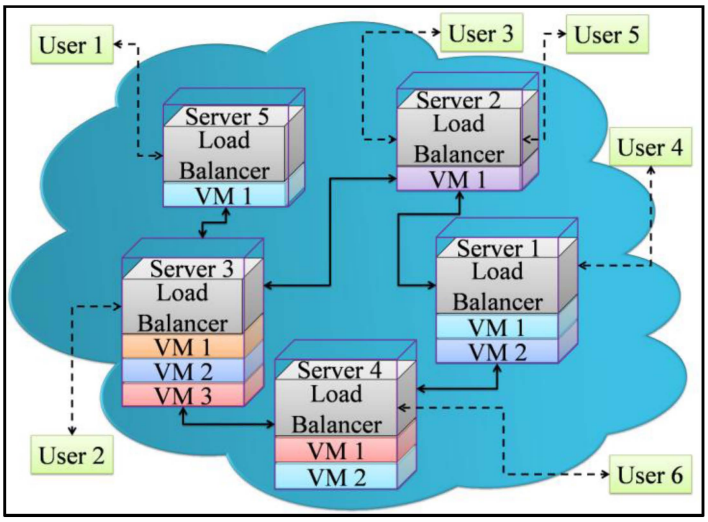

Fig. 1. Load balancer in cloud.

In projects like Cloud@Home [2], the resources are shared and dynamic in nature. This adds additional burden to the load balancing algorithm. For Platform as a Service (PaaS), the cloud users post their jobs into the cloud that will be executed in the cloud and result will be sent as a response to the user. Here wait time in execution plays a vital role in user experience. Also over burden on server costs in cooling of the system, power consumption, and increased number of job failures. This paper

978-1-4799-6085-9/15/$31.00©2015 IEEE

proposes a load balancing algorithm that focuses on reducing wait time and evenly distributing the load.

B. Related work

Plenty of algorithms are available for load balancing in cloud computing such as round-robin, ant colony and honey bee inspired algorithms, biased random sampling, Max-Min and Min-Min. Round-robin, Max-Min and Min-Min are designed for static environment. Also they require a centralized system for execution; hence they are incapable to address the network dynamics.

Fundamental concept of honeybee or ant colony inspired load balancing algorithms [3, 4, 5] is to employ pheromone trails of ants or waggle dance of honey bees as an indirect means of communication. In honey bee systems, the forager expresses the quality and quantity of newly discovered food source by waggle dance and harvesters follow the forager to collect the honey. Upon returning, harvesters inform the details of remaining food by waggle dance. During load balancing operation of honeybee inspired algorithm, the servers take the role of forager with probability p_x. These dedicated foragers explore other servers and exploit the resources. Every server posts on the advert board with probability p_r after completing the request. Any server that reads advert follows it and mimics harvester behaviour; server that does not reads the advert will behave as forager [3]. The dedication of server costs in terms of resources and dynamics of topology makes the advert unreliable.

Biased random sampling algorithm [4, 6, 7] uses virtual graphs for load balancing. This algorithm was originally developed for grid computing. When a job is received, a random walk is performed. The walk starts at the node that receives the job and proceeds with its randomly selected neighbour. The number of nodes covered in the walk is limited by a threshold value. The last node of the walk executes the job. Although this method distributes the load due to randomness, it is highly possible to have a situation where one server is heavily loaded while the others are underutilized.

An improved version of the random sampling algorithm [1] selects least loaded node from the random walk for the execution of job. This algorithm uses queue length or remaining execution time as cost to compute load of a node. This cost function helps in reducing the chance of having overloaded nodes while other nodes are idle, yet the randomness of selection process may lead to such possibilities.

C. Contribution of the paper

The contribution of this paper for load balancing is listed below:

- An equation to estimate load of a neighbour based on known information about the neighbour is given in Eq. (1).
- An equation to predict the current load of a neighbour node based on the estimation from Eq. (1) is given in Eq. (2).
- Two strategies are proposed to select a neighbour node for random walk.

The rest of the section is organized as follows. Section II describes the assumption, random walk procedures and the proposed strategies. Section III explains the algorithm and its implementation details. The simulation results can be found in Section IV and Section V concludes the work.

II. Design Overview

A. Assumptions

Neighbour aware random walk algorithm is designed with the following assumptions:

- Every node in the network is aware of the number of nodes in the network (an approximate estimation) and their neighbours.
- Each node has different number of virtual machine. Each virtual machine has equal computational resources.
- The duration of resource requirement is known.
- Each physical machine knows the number of virtual machines running in their neighbour.

These assumptions are taken from [1]. Although the knowledge about exact number of nodes in the network will improve the performance of random walk, NARS gives flexibility to know only a fair approximation of the number of nodes in the network. This gives the freedom for nodes to join and leave the network at any time.

B. Random walk procedure

When the load balancer of a node receives a job request, it sends a *RequestToken* with the duration of resource request and load of current node to a randomly selected neighbour. The node that receives the *RequestToken* adds its own load and forwards to a randomly selected neighbour until the number of nodes covered by the *RequestToken* is equal to the walk threshold of the network, which is $\lceil \ln(N) \rceil$ for an N node network.

The last node to receive the *RequestToken* selects the least loaded node from the *RequestToken* and allocates the job to that node. The last node in the walk sends *AllotmentToken* to the node which is selected for the job. Upon receiving the *AllotmentToken*, the allotted node can receive the job details such as inputs of the job, executable and other particulars from the node which initiated the random walk.

C. Proposed Strategies

In neighbour aware load balancing, each node plays a game in which the node tries to predict the current load of its neighbours based on previous knowledge and selects the least loaded node for the random walk. The knowledge acquisition

can be done in two ways:

1. When a node receives a *RequestToken*, it gathers load information of every neighbour that has participated in that walk and updates the knowledge base with timestamp.
2. When a node A relays the *AllotmentToken* or job details to node B and B is neighbour of A, then the load of that job is added to the known load of B in A's knowledge base.

This paper proposes two strategies to select a neighbour for the random walk. The node estimates the current load of its neighbour based on current time, number of resources available at that neighbour, known load of that neighbour and the timestamp of the knowledge using Eq. (1). Then it generates the probability P_i for each neighbour based on Eq. (2).

$$L_i = \left(\frac{l_i}{R_i}\right) - [(t_c - t_i) \times R_i] \; \forall i = 1, 2 \ldots n. \qquad (1)$$

Where,

L_i – Estimated load of i^{th} neighbour.
l_i – known load of i^{th} neighbour.
R_i – number of resources available at i^{th} neighbour.
t_c – current timestamp.
t_i – timestamp of recent update from i^{th} neighbour.
n – Total number of neighbours.

$$P_i = \left(\sum_{j=1}^{n} L_j\right) - L_i \Big/ \left(\sum_{j=1}^{n} L_j\right) \times n \quad \forall i = 1, 2 \ldots n. \qquad (2)$$

Where,

P_i–Probability that i^{th} neighbour is least loaded.
L_i – Estimated load of i^{th} neighbour using Eq. (1).
n– Total number of neighbours.

1) Strategy 1,Probability based: A random number based on the calculated probabilities (using Eq. (2)) is generated which is used to choose the node for next step of walk. This can be done by constructing a scale *S* from 0 to 1 and assign range for each node based on their probability (i.e. the first node in the neighbour list gets range from 0 to P_0 and second node gets range from P_0 to P_0+P_1 and so on). A number r ($0 \leq r \leq 1$) is generated using random number generator, and the neighbour whose *S* range covers *r* is selected for random walk.

2) Strategy 2, Least loaded node first: In this strategy, the node estimates the load of each neighbour based on the knowledge it gathered from participating in prior walks and selects the least loaded node. Here the randomness of the neighbour selection depends on the walks in which the node has participated.

Both strategies ensure that the node explores the least loaded neighbour and keeps the fresh data about those neighbours. In this process it is possible that the details of heavily loaded neighbours are not updated and there may be change in topology. Since those neighbours are loaded with previously assigned jobs, the impact of such topological change does not affect the robustness of the load balancing process.

III. Algorithm Design and Implementation

A. Neighbour aware random sampling algorithm

When a node (*K*) in the network receives a job (*J*) from user, it creates a *RequestToken* (*RT*) for the job. Particulars of the node *K* and details of the job *J* are added to the request token *RT*; the *walkLength* of the token *RT* is initialized to 1. Then the node calls the *selectNeighbour* function to select a random neighbour from its neighbour list *NL* based on the version of algorithm used (*ver*). After receiving the ID of random neighbour node (*M*), the request token *RT is* sent to the node *M*. The sequence diagram of the procedure to be followed when receiving a job is shown in Fig. 2.

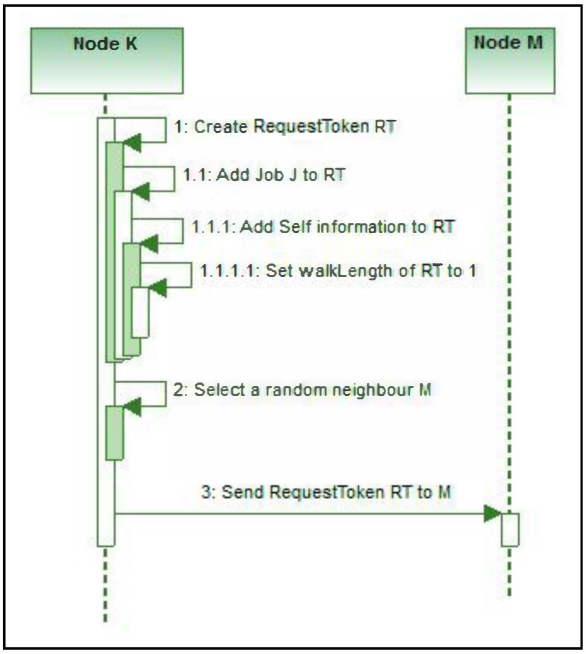

Fig. 2. Sequence diagram of Job reception and *RequestToken* generation.

Upon receiving a request token *RT,* the node *M* checks if the *walkLength* is less than the *threshold* value. If the *walkLength* is less, then it adds its details to the request token *RT*, increments *walkLength*, updates its local cache with the details available from the token *RT*, and selects a neighbour using *selectNeighbour* method and forwards the token to that neighbour. When *walkLength* reaches *threshold*, the node *M* searches for the least loaded node from the request token *RT*, creates an *AllotmentToken* and adds job details to the token and sends the *AllotmentToken* to the least loaded node, which will execute the job *J*. The flow chart of the algorithm can be seen in Fig. 3.

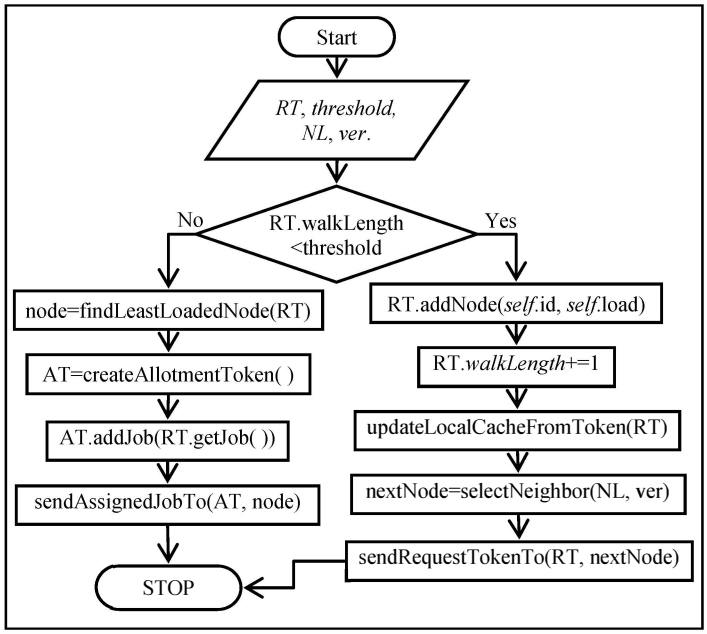

Fig. 3. Flow chart of *RequestToken* processing.

The algorithm for selecting neighbour (*selectNeighbour*) returns a node ID from the neighbour list *NL* for random walk. In case of *LeastLoaded* version of the algorithm, *selectNeighbour* calls *leastLoaded*( ) function, which returns the minimum loaded neighbour's ID by applying Eq. (1) over the knowledge acquired from previous walks. In case of probability based algorithm, the node computes the probability P_i for each neighbour. Then a random number r between 0 and 1 is generated. The probability of each neighbour is mapped from 0 to 1 to construct a scale S. The function *selectProbInRange*( ) returns the index of the neighbour whose range in S covers the random number r. Pseudo code for neighbour selection algorithm is given in Algorithm 1.

Algorithm 1: Pseudo code for neighbour selection

```
selectNeighbor( ):
    Input:
        NeighborList NL
        AlgoVersion ver
    Output:
        Node node
Proc:
        If (ver==LeastLoaded):
                node = leastLoaded(NL)
        Else:
                P = computePi(NL)
                r = genRandomNumber(0, 1)
                S = constructScale(P)
                i = selectProbInRange(r, S)
                node = NL.getNodeByIndex(i)
        EndIf
        Return node
    EndProc
```

B. *Implemetation details*

Both versions of the neighbour aware random sampling algorithm are implemented with execution time as cost for load computing. First In First Out (FIFO) order is followed to execute the jobs assigned to a server. The improved version of random sampling algorithm [1] is also implemented with execution time as cost to compute load. The implementation has been done in JAVA and a dedicated cloud environment is created to observe the precise effect of load balancing algorithm and isolate delays from other variables of cloud network. The results are analysed in Section IV.

IV. Results and Analysis

Improved biased random sampling algorithm and NARS are simulated on virtual cloud running over a Windows 7 system with Intel Core i3 processor @ 3.30GHz and 4GB RAM. The cloud consists of twenty nodes with varying number of virtual machines ranging from 2 to 6. To make a fair analysis, fifty trials were done to avoid the effect of infrequent outcome due to randomness of the algorithm.

A. *Average waiting time*

Wait time is the time interval between the arrival of job to a virtual machine to the completion of its execution. If an algorithm can reduce the average wait time of jobs, then it means the algorithm is capable of exploring the least loaded nodes and hence reduces the wait time. The average waiting time of jobs for various number of jobs are shown in Fig. 4.

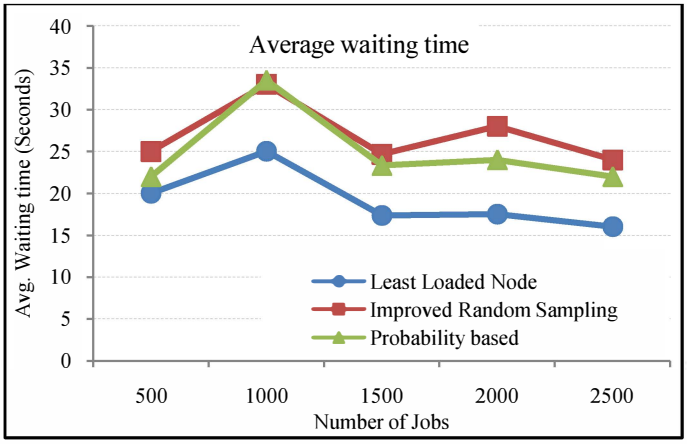

Fig. 4. Average waiting time.

It can be observed that 'Least Loaded Node' strategy's performance is better compared to other methods due to its ability to keep wait time in control. In case of probability based strategy or improved random sampling, the involvement of randomness increases the wait time in some scenarios by choosing the loaded nodes for random walk. The increase in number of jobs escalates the probability of exploring other nodes, hence the wait time reduces.

B. *Maximum wait time*

A user whose job experiences more wait time is unlikely to use the service again. Hence, maximum wait time of a process is also taken as a consideration for the analysis. The maximum wait time of the three algorithms can be seen in Fig. 5.

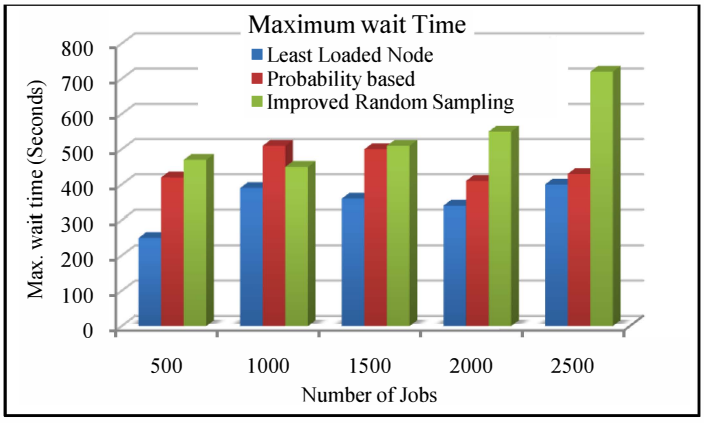

Fig.5. Maximum wait time

C. *Standard deviation of load per node*

A node failure causes more resource consumption due to re-execution of jobs. To rectify this, the load can be shared equally among the nodes which cause minimum over head in case of failure in dynamic environments. So standard deviation of load per virtual machine is taken as a measure for comparison. Ideally standard deviation per node is expected to be zero for uniform distribution of load among all nodes on the network. Due to randomness of node selection and dynamic environments, some nodes may be loaded more when compared to others. Fig. 6 shows the standard deviation of load per virtual machine. It can be seen that the least loaded algorithm distributes load uniformly when compared to the other two.

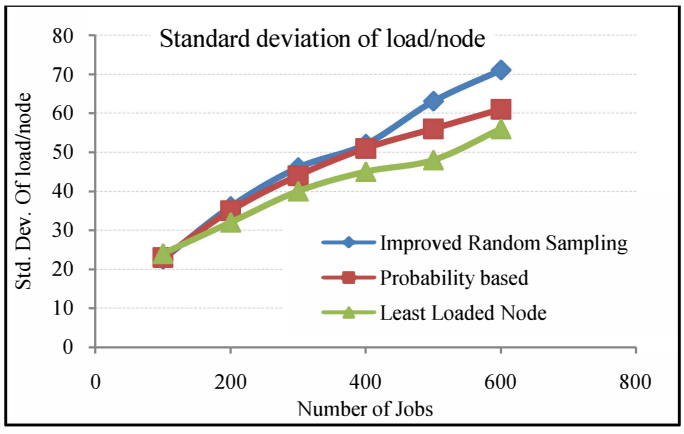

Fig.6. Standard Deviation of load per VM

V. CONCLUSION

Neighbour aware load balancing algorithm is proposed in this paper. Two strategies for selecting neighbour for random walk have been discussed. From the simulation, it is clear that selecting least loaded node for random walk performs better than probability based selection or the improved random walk algorithm. Over a large dense network, least loaded neighbour of one node may get loaded by some other nodes and become heavily loaded node. In such cases probability based strategy allows the nodes to explore other nodes and enhance the knowledge about new nodes. Hence it improves the probability of assigning jobs to lightly loaded nodes.

REFERENCES

[1] Sheeja S Manakattu, S. D. Madhu Kumar, "An improved biased random sampling algorithm for load balancing in cloud based systems, Proceedings of the International Conference on Advances in Computing, pp 459-462, 2012.

[2] Vincenzo D. Cunsolo, Salvatore Distefano, Antonio Puliafito, Marco Scarpa, "Volunteer Computing and Desktop Cloud: The Cloud@Home Paradigm", IEEE International Symposium on Network Computing and Applications, pp 134-139, 2009.

[3] Sunil Nakrani and Craig Tovey, "On honey bees and dynamic server allocation in Internet hosting centers", Adaptive behaviour – Animals, Software Agents, Robots, Adaptive Systems, pp. 223-240, September 2004.

[4] Martin Randles, David Lamp, and A. Taleb-Bendiab, "A comparative study into distributed load balancing algorithms for cloud computing", Advanced Information Networking and Applications Workshop, pp. 551-556, 2010.

[5] R. Rastogi Kumar Nishant, Pratik Sharma, "Load balancing of nodes in cloud using ant colony optimization", Proceedings of the 14th international conference on computer modeling and simulation (UKSim), IEEE, pp 3-8, March 2012.

[6] O. A. Rahmeh, P. Johnson, and A. Taleb-Bendiab, "A dynamic biased random sampling scheme for scalable and reliable gird networks", INFOCOMP journal of computer science, pp 01-10, 2008.

[7] M. Randles, O. Abu-Rahmeh, P. Johnson, and A. Taleb-Bendiab, "Biased random walks on resource network graphs for load balancing", Journal of SuperComputing., Vol 54, pp 138-162, July 2010.