Load Balancing Algorithm for Better Response Time using Improved Queuing (BRIQ)

Dibyadarshan Hota[1], Aditi Gupta[2], Annappa .B[3]
Department of Computer Science and Engineering
National Institute of Technology, Karnataka
Surathkal, Mangaluru, Karnataka, India
dibyadarshanhota@gmail.com[1], guptaaditi1709@gmail.com[2], annappa@ieee.org[3]

***Abstract*—Cloud computing has become a prominent field in Computer Science since it aids in on-demand availability of computer system resources. However, ensuring that the workloads are distributed efficiently across the servers, is a challenging task. The class of algorithms which assist in this task are known as load balancing algorithms. Efficient distribution is necessary to avoid overloading of servers, improving resource utilization, reducing latency, and maximizing the throughput. Many load balancing algorithms exist in the literature, such as, round robin, least connection, agent-based adaptive load balancing, weighted response time, etc. However, all of these algorithms have certain short-comings which makes them ineffective in a lot of real-world scenarios. Most of the state-of-the-art algorithms do not account for the actual instant power of virtual machines. We propose an algorithm which overcomes this drawback and improves the response time and processing time.**

***Index Terms*—Cloud Computing, Distributed Systems, Load Balancing, Scheduling Policy, Virtual Machines, Data Migration**

I. Introduction

Load balancing is a technique of thoughtfully offloading the distribution of workload among the connected nodes in a parallel and/or distributed computing environment to improve the performance of the system. Load balancing is necessary to ensure optimal resource utilization and absence of overloaded servers. Load balancing can be broadly classified into *two* types - *static* and *dynamic*.

While static load balancing does not take the state of the system (such as, node performances) into account for making decisions about distribution of the task, dynamic load balancing does.

Static load balancing algorithms work with a priori knowledge about the services and their overall status, resource requirements and communication time. They are preferable in scenarios where the behaviour of job queue is predictable and tasks are fairly similar in nature. They have low processing time and are less complex than dynamic load balancing algorithms.

Dynamic load balancing algorithms, on the other hand, react spontaneously to the changes occurring in the system. They usually follow a master-slave architecture wherein the master assigns incoming processes to the slaves based on the state of the environment.

Load balancing is performed at *two* levels - host level (known as *Virtual Machine (VM) scheduling policy*) and VM level (known as *task scheduling policy*). Furthermore, decisions can be made at either level to pick between *time-shared (TS)* scheduling and *space-shared (SS)* scheduling.

Chien et al. (2016) [1] proposes a dynamic load balancing algorithm based on estimating the finish times of the jobs. It builds upon Active Monitoring Load Balancer (AMLB) algorithm proposed in [2]. AMLB maintains a datacenter broker which logs information and current activity status of each VM. Upon the arrival of a job, it identifies the least loaded VM and assigns the job to it. Although the algorithm improves the response time and works fine to reduce bottleneck scenarios, it does not take into account the job size and actual instant powers of VMs. To counter this issue, Chien et al. proposes an algorithm to take both the missing factors into consideration. It demarcates four scheduling scenarios as follows and aims to improve the processing time and response time:

1) SS in host level and SS in VM level
2) SS in host level and TS in VM level
3) TS in host level and SS in VM level
4) TS in host level and TS in VM level

The algorithm utilises a datacenter broker to maintain the status of all VMs. While there is a job available in the queue to be allocated, it calculates the finish time of the job for each VM using the formulae discussed later. Finally, it picks the VM with the earliest completion time and assigns the task to it.

The proposed algorithm has a shortcoming that although it considers the earliest finish time which can be offered to a client request by a virtual machine, it fails to schedule the request within the virtual machine queue effectively. It stores the incoming jobs in a queue and processes them in a First In, First Out (FIFO) fashion. This does not allow much room to decide upon as to which job should be processed next.

This paper [1] aims to improvise the algorithm by proposing a heuristic to decide the next job to be allotted in order to reduce the net response time. This is achieved by storing the incoming jobs in a priority queue (heap) and using ageing to prevent starvation.

[1]Code to reproduce our results can be found here: https://github.com/aditigupta17/load-balancing-briq/

II. Related Work

The following section analyses some of the load balancing algorithms.

A. Round Robin

Round robin load balancing is effective when all the servers have similar compute capabilities, specifications and storage capacity. In this, the load balancer assigns the client requests to the servers in a specific sequence. Upon reaching the end of the sequence, it loops back to be beginning, i.e. client requests are handled in a cyclic manner.

The main advantage of this algorithm is that it is effective if servers are homogeneous in nature, and it is extremely easy to implement such a load balancer. However, it is common to have heterogeneous systems in place, in such cases the following variants are useful:

- Weighted Round Robin: Each of the servers is assigned a weight depending on the load-handling capacity of the server, decided using an adaptive algorithm or by the system administrator. A server with higher weight is assigned a larger portion of the client requests.
- Dynamic Round Robin: The servers are assigned weights dynamically depending on the server's current capacity and idleness. Then, the client requests are assigned to a server, proportional to its weight.

B. Least Connection

A least connection load balancer assigns a client request to the server which has the least number of active connections. This is a dynamic load balancing algorithm since it makes use of the server's current state, rather than following an arbitrary sequence, unlike round robin approaches. The main disadvantage of such an approach is that the current server load is not taken into consideration while distributing requests, this leads to degradation of performance of the load balancer.

There exists a weighted variant for least connection load balancing algorithm as well. In weighted least connection load balancing, a numerical weight is assigned to each server depending on the compute capability of the server. If multiple servers have the same number of active connections then the server with higher weight is chosen to process the request.

C. Mapping Policy Based Load Balancing

Junjie Ni et al. [3] proposes a load balancing algorithm based on virtual machine to physical machine mapping. The algorithm contains a resource monitor and a scheduling controller. The resource monitor tracks the usage, idleness, and availability of the resources. Whereas the scheduling controller does the task of assigning requests to the appropriate server.

D. Load Balancing Using Ant Colony Optimization

K. Nishant et al. [4] proposes a probabilistic load balancing algorithm which makes use of ant-colony optimization (ACO). In the approach proposed by the paper, the ants (multi-agent, inspired by behaviour of real ants) originate from the head node and traverse the entire network. While traversing, they keep track of overloaded and underloaded nodes in the network, and the pheromone table is updated. The pheromone table gives an estimate of resource utilization by each node.

The traversal is done using two types of movements:

- Forward movement: The ants continuously move in the forward direction in the network, and keep tack of overloaded and underloaded nodes.
- Backward movement: If an ant encounters an overloaded node and it has previously encountered a underloaded node, it redistributed the work. The process can be vice-versa as well.

III. Proposed Algorithm

In the proposed algorithm, we assign the incoming client request to the virtual machine such that estimated finish time of the service is minimum. To do this, we calculate *finishTime* according to formulae *(2)* and *(3)* for all the virtual machines. The VM which gives the minimum *finishTime* is assigned the incoming client request.

Once a client request is assigned to a VM, it gets added to the *virtualMachineQueue*. The *virtualMachineQueue* behaves like a priority queue, it gives priority to the task which has the least *instructionCount*. This ensures that the *NetResponseTime (9)* of the load balancer is minimized.

Response time for a job with arrival time *arrivalTime*, finish time *finishTime* and transmission delay *TDelay* is given as:

$$ResponseTime = finishTime - arrivalTime + TDelay \tag{1}$$

Assuming *startTime(x)* as the time when the job *x* started, *time* as the current simulation time, *instructionCount* as the total number of instructions of a job, *core(x)* as the number of cores or processing elements required by the job, *capacity* as the average processing capacity (in MIPS) of a core for a job, we obtain following equations:

If the host scheduling policy is SS-SS or TS-SS, *finishTime* is calculated as:

$$finishTime = startTime + \frac{instructionCount}{capacity * core(x)} \tag{2}$$

If the host scheduling policy is SS-TS or TS-TS, *finishTime* is calculated as:

$$finishTime = time + \frac{instructionCount}{capacity * core(x)} \tag{3}$$

Execution time of a job is calculated as:

$$ExecutionTime = \frac{instructionCount}{capacity * core(x)} \tag{4}$$

Capacity in each case is calculated as follows:

1) VM scheduling policy is SS, task scheduling policy is SS:

$$Capacity = \sum_{i=1}^{np} \frac{Cap(i))}{np} \tag{5}$$

where, *Cap(i)* is the processing power of the core *i*, *np* is the number of real core of the considered host.

2) VM scheduling policy is SS, task scheduling policy is TS:

$$Capacity = \frac{\sum_{i=1}^{np} Cap(i))}{max(\sum_{j=1}^{\alpha} cores(j), np)))} \quad (6)$$

where, *cores(j)* is number of cores that job *j* needs. α is total job in VM which contains the job.

3) VM scheduling policy is TS, task scheduling policy is SS:

$$Capacity = \frac{\sum_{i=1}^{np} Cap(i))}{max(\sum_{k=1}^{\beta} \sum_{j=1}^{\gamma} cores(j), np)))} \quad (7)$$

where, β is the number of VMs in the current host, γ is number of jobs running simultaneously in VM_k.

4) VM scheduling policy is TS, task scheduling policy is TS:

$$Capacity = \frac{\sum_{i=1}^{np} Cap(i))}{max(\sum_{j=1}^{\delta} cores(j), np)))} \quad (8)$$

where, δ is total job of the considered host.

However, this approach creates a problem, if a task has extremely low priority it is likely that it will never get scheduled. This problem is commonly known as starvation. The proposed algorithm resolves this issue by using *ageing*. In *ageing* a client request's priority is artificially increased by an amount quantified by its wait time. This ensures that upon passage of time, even a request with low *instructionCount* will eventually gain sufficient priority to get scheduled. Such an approach guarantees low response time and no starvation.

The *NetResponseTime* for the load balancer is finally calculated using *ResponseTime* of each request as follows:

$$NetResponseTime = \sum_{i=1}^{requestCount} \frac{ResponseTime(i)}{requestCount} \quad (9)$$

IV. Experimental Results

Significant improvements have been achieved with respect to response time and throughput. Metrics have been logged in the tables below. For the purpose of simulation, a variable number of cores were used (ranging upto 5) per VM. The number of client requests were varied by a factor of 10, starting from 10, as indicated in the tables. The algorithm was implemented using C++ (gcc version 7.5.0). The experiments were run on a system having Intel® Core™ i7-8550U CPU @ 1.80GHz × 8 processor.

TABLE I
Response Time: Load Balancing using Estimated Finish Time Algorithm versus BRIQ

Number of Client Requests	*Estimated Finish Time Algorithm*	*BRIQ*
10	14.944537	14.944537
10^2	38.853931	36.879389
10^3	434.767279	313.244709
10^4	4549.779616	3205.598167
10^5	45868.511618	32449.084248

*Response Time reported in milliseconds

TABLE II
Throughput: Load Balancing using Estimated Finish Time Algorithm versus BRIQ

Number of Client Requests	*Estimated Finish Time Algorithm*	*BRIQ*
10	0.05614448	0.05614448
10^2	0.02767749	0.03199268723
10^3	0.00228050171	0.00396647467
10^4	0.00021972473	0.0003918567
10^5	0.00002177804	0.0000384324

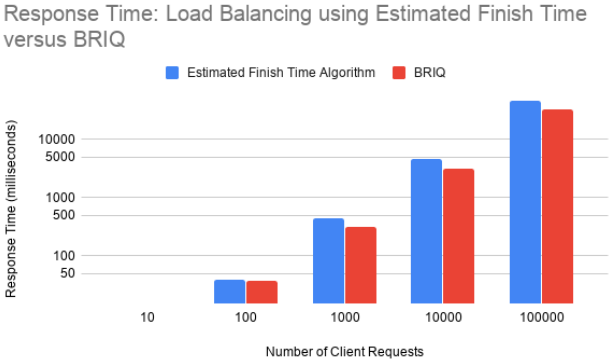

Fig. 1. Response Time: Load Balancing using Estimated Finish Time Algorithm versus BRIQ

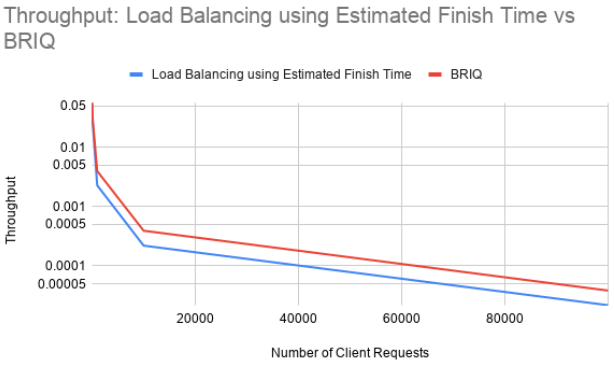

Fig. 2. Throughput: Load Balancing using Estimated Finish Time Algorithm versus BRIQ

V. Conclusion

Load balancing assists in efficient utilization of cloud computing resources and also affects the issue of data center power consumption. The paper proposes an efficient load balancing algorithm based on the method of estimating the end of service time combined with a *shortest job first* queue, within the virtual machines. It is accompanied by *ageing* to prevent starvation in heterogeneous cloud computing environments. We considered scheduling cases of space sharing (SS) in host level and space sharing (SS) in VM level, a set of formulae were developed to calculate the average response time and throughput of the virtual core. Simulation results showed that the proposed algorithm is more effective, processing time and response time are improved as compared to other state-of-the-art approaches. Such an efficient algorithm will lead to better usage of computational resources.

References

[1] N. K. Chien, N. H. Son and H. Dac Loc, "Load balancing algorithm based on estimating finish time of services in cloud computing," *2016 18th International Conference on Advanced Communication Technology (ICACT)*, Pyeongchang, 2016, pp. 228-233.

[2] Jasmin, J., Bhupendra, V., "Efficient VM load balancing algorithm for a cloud computing environment", *International Journal on Computer Science and Engineering (IJCSE)*, 2012.

[3] J. Ni, Y. Huang, Z. Luan, J. Zhang and D. Qian, "Virtual machine mapping policy based on load balancing in private cloud environment," *2011 International Conference on Cloud and Service Computing*, Hong Kong, 2011, pp. 292-295.

[4] K. Nishant et al., "Load Balancing of Nodes in Cloud Using Ant Colony Optimization," *2012 UKSim 14th International Conference on Computer Modelling and Simulation*, Cambridge, 2012, pp. 3-8.

[5] T. Deepa and D. Cheelu, "A comparative study of static and dynamic load balancing algorithms in cloud computing," *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, Chennai, 2017, pp. 3375-3378.

[6] Lu Kang and Xing Ting, "Application of adaptive load balancing algorithm based on minimum traffic in cloud computing architecture," *2015 International Conference on Logistics, Informatics and Service Sciences (LISS)*, Barcelona, 2015, pp. 1-5.

[7] K. D. Patel and T. M. Bhalodia, "An Efficient Dynamic Load Balancing Algorithm for Virtual Machine in Cloud Computing," *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, Madurai, India, 2019, pp. 145-150.

[8] L. Zhu, J. Cui and G. Xiong, "Improved dynamic load balancing algorithm based on Least-Connection Scheduling," *2018 IEEE 4th Information Technology and Mechatronics Engineering Conference (ITOEC)*, Chongqing, China, 2018, pp. 1858-1862.