A Dynamic Load Balancing Model Based on Negative Feedback and Exponential Smoothing Estimation

Di Yuan, Shuai Wang, Xinya Sun
Tsinghua University
Beijing, 100084, China
{yuan-d09, wangshuai04}@mails.tsinghua.edu.cn, xinyasun@tsinghua.edu.cn

***Abstract*—Server clusters can be used to manage the massive number of requests that a hot website will receive, so as to meet the rapid development of Internet application. The Linux Virtual Server provides a good solution for cluster revision, and there is software that can be used for management and monitoring. However, the scheduling algorithms of Linux Virtual Server are not sufficient to deal with the heavy load balancing required today. A dynamic load balancing scheduling algorithm has been proposed to solve the problems of static algorithms, but we find that there are some drawbacks in actual use. In this paper, we suggest an improved dynamic load balancing model that overcomes the limits or drawbacks of the simple dynamic algorithm. In the suggested model, negative feedback and exponential smoothing estimation methods have been used to improve the load balancing effect. Besides, service response time has been used to adjust the weight variation to achieve better effect. The suggested model is implemented in our dynamic load balancing algorithm. Experiments show that, our algorithm can achieve better performance than the existing static and dynamic algorithms.**

Keywords-load balancing; dynamic algorithm; negative feedback; exponential smoothing estimation; throughput

I. INTRODUCTION

With the rapid development of the Internet, hot web sites must cope with greater demands than before. Increasing number of users or clients makes a single server not sufficient to handle this aggressively increasing load. As a result, we ought to use a server cluster to solve this problem. A server cluster can help to keep computing service in good quality by adjusting the server nodes dynamically when the number of requests suddenly changes. However, we should find a method to assign the new connections to the computing elements properly.

Server cluster can be built with either expensive hardware as F5 load balancer, or Linux Virtual Server (LVS) [1]. LVS is a good solution to companies for cost factors. Generally, the LVS is a software tool assigns connections to multiple servers, which can be used to build highly scalable and highly available services. An LVS cluster is composed by the load balancer and real server nodes. The load balancer receives requests and schedules them to real servers following certain rules [2].

The LVS clusters are always built by Direct Routing method, because load balancer is independent from OS and the load balancer's burden is less than server nodes [3]. LVS has ten scheduling algorithms [4]. The WLC algorithm, which schedules the new connections according to servers' weights and number of active connections, is most commonly used for its good balancing performance [3]. However, it is usually difficult to locate proper weight to a server, and the weight can only be adjusted manually while LVS is running. Moreover, if the requests vary in their processing time or package size, the workload of servers will be skewed.

A basic dynamic load balancing algorithm based on negative feedback has been proposed [5]. Daemon tools like *Keepalived* or *HeartBeat* can be used to manage server nodes. The load balancer collects load information of a server node, which can be used to update its weight through the Simple Network Management Protocol (SNMP). Aggregated load of a server node can be calculated by load information, and new weight of the server node can be solved by

$$W_i = W_{i-1} + W_{\text{step}} \sqrt[3]{A - \text{Aggregate\_Load}_i} \qquad (1)$$

In the above equation, 'W_{step}' denotes the step of weight adjustment, while 'A' denotes the expected value of the aggregate load. Through analysis and experiments, we have found this dynamic algorithm have drawbacks in actual use. Firstly, the current weight 'W_i' only relies on 'W_{i-1}' and the current aggregate load. Once the collection or calculation of load is interfered, the adjustment of weight may not reflect actual variation of load. Secondly, the response time of the server, an important factor of server's load, is aggregated to the 'Aggregate_Load', which may undermine its importance. Lastly, no matter how much the variation of aggregate load is, the upper bound of the weight adjustment is 'W_{step}'. As a result, the update of the weight has limitations, which may affect the load balancing effect.

In this paper, we suggest a new load balancing model to improve the load balancing performance. We set up a cluster system with the characteristics of high availability and high reliability based on LVS and open source software *Keepalived*, in order to implement our model and algorithm. The load balancer checks server nodes by using *Keepalived*, and collects the real-time load information through a user-defined monitoring module. Then new weights of the server nodes are calculated through weight evaluation module with the load information and updated into Linux kernel. The load balancer assigns new connections by using weighted

 ISBN: 978-1-61208-187-8

scheduling algorithm of LVS according to new weights [6]. In weight evaluation module, we collect load information of the server nodes and evaluate the aggregated load. Besides, the module detects response time from each server node to correct 'W_{step}'. Furthermore, the module calculates weight estimation through exponential smoothing method, the purpose of which is to make the adjustment of weight consistent with the actual variation of load. We suggest an improved dynamic load balancing algorithm based on improvements above and do experiments through open source software *Apache JMeter* [7]. The new algorithm shows better result of balancing effect than the existed WLC algorithm and simple dynamic algorithm above.

The remainder of this paper is organized as follows: Section II is a focus of this paper, our dynamic load balancing model is described. In Section III, framework and flow of corresponding algorithm is presented. In Section IV, experiments of three algorithms are done to compare the balancing performance. In Section V, some conclusions are drawn through the experiments.

II. DYNAMIC LOAD BALANCING

A. Negative Feedback Model of Dynamic Load Balancing

The WLC algorithm schedules the new connections according to weights and number of active connections. As the former factor is static during the scheduling process, WLC is essentially a static scheduling algorithm [8]. By contrast, our dynamic load balancing algorithm schedules the connections according to both active connections number and load information. The load balancer sends request to server nodes to get load information, and then the weight evaluation module calculates new weight according to former weights and aggregated load. The load balancer schedules the new connections from client to server nodes according to new weights. It is obvious that the dynamic load balancing is a negative feedback process.

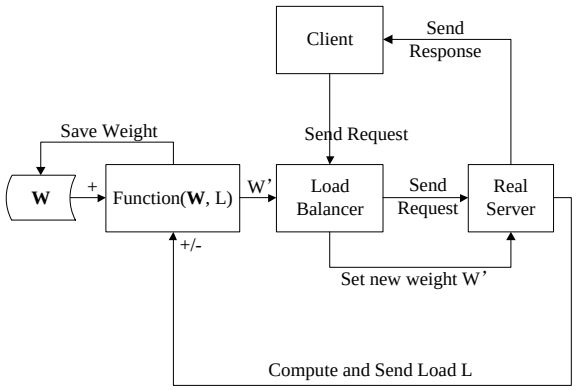

Figure 1. Negative feedback model of dynamic load balancing

In order to compute the weight of the server node, load information collection service is running in each server node. The load balancer collects load information 'L' periodically. Weight evaluation module 'Function(**W**, L)' calculates the new weight by former weight vector '**W**' and aggregated load 'L' and then update the IPVS scheduling table. This dynamic algorithm can overcome the drawbacks of WLC, and the effect of load balancing will be enhanced [9].

B. Weight Evaluation Module

As we discussed above, the weight evaluation module of load balancer is an important part of this dynamic algorithm. The load information can be used to calculate new weight of the server.

Assume vector $\mathbf{L}=[L_1, L_2, L_3, L_4]$, ($L_i<1$) denotes the load parameters and vector $\mathbf{Q}=[Q_1, Q_2, Q_3, Q_4]$ denotes proportionality factor of the parameters, where L_1 to L_4 represent CPU usage rate, memory usage rate, file system usage rate, and one server's new connections proportion of the total number. Thus, $0<\mathbf{QL}^{\mathrm{T}}<1$, the aggregated load parameter, denotes the current load of a server node [10]. Further, we assume that T_{delay} denotes the real response time of the computing service and T_{ideal} denotes the ideal value of T_{delay}, then the ratio of them may represent the current network state between load balancer and the server node.

The basic dynamic load balancing model assumes that, current weight of a server node is only related to the former one [5]. Considering the fact that the aggregated load may be interfered, this mechanism may lead to deviation of the weight calculation. Our improved dynamic algorithm solves this problem through three former weights. Assume vector $\mathbf{W} = [W_{i-2}, W_{i-1}, W_i]$ denotes weights before time i+1 and vector $\mathbf{P} = [P_1, P_2, P_3]$ denotes proportionality factor of the weights. As we discussed above, estimated weight at time i+1 should be used to adjust the 'W_{step}', in order to make the step more proper. Assume 'A' denotes the expected value of the aggregate load, the weight update formula is

$$W_{i+1} = \mathbf{PW}^{\mathrm{T}} + W_{step}\left(\frac{\hat{W}_{i+1}}{\mathbf{PW}^{\mathrm{T}}}\right)\left(\frac{T_{ideal}}{T_{delay}}\right)^{\mathrm{sgn}\left(A-\mathbf{QL}^{\mathrm{T}}\right)}\sqrt[3]{A-\mathbf{QL}^{\mathrm{T}}} \qquad (2)$$

For each server node, *Keepalived* may set its weight from 1 to 253 [6]. We can set the weight range $[w_0, 10w_0]$, ($0<w_0<25$) for simplicity. The ideal service response delay T_{ideal} can be estimated through experiments. In order to properly reflect the impact of service response time, we set $T_{ideal}<T_{delay}<1.5T_{ideal}$. If the aggregated load $\mathbf{QL}^{\mathrm{T}}$ is greater than A, weight adjustment and T_{delay} is proportional, and vice versa. W_{step} and A are two important parameters. Generally, we set $W_{step}= w_0/2$ and $0.45<A<0.95$. As there must be some differences between different cluster systems, the exact value of them should be determined through experiments [9]. We set W_0 the reference value of W_1 to W_3, and then the complete weight evaluation module is

$$W_{i+1} = \begin{cases} W_0 + W_{step}\dfrac{T_{delay}}{T_{ideal}}\sqrt[3]{A-\mathbf{QL}^{\mathrm{T}}} & i<3 \\ \mathbf{PW}^{\mathrm{T}} + W_{step}\left(\dfrac{\hat{W}_{i+1}}{\mathbf{PW}^{\mathrm{T}}}\right)\left(\dfrac{T_{ideal}}{T_{delay}}\right)^{\mathrm{sgn}\left(A-\mathbf{QL}^{\mathrm{T}}\right)}\sqrt[3]{A-\mathbf{QL}^{\mathrm{T}}} & i\geq 3 \end{cases} \qquad (3)$$

C. Exponential Smoothing Estimation

As discussed above, new weight should be estimated to adjust 'W_{step}' to make the adjustment of weight consistent with the actual variation of load. We estimate the new weight through linear quadratic exponential smoothing method, the essence of which is to get the estimation result through the weighted average of historical data. According to exponential smoothing theory, time series trend with stability or regularity, so they can be reasonably extended to estimate the future trend [11]. Exponential smoothing method, mainly used for variable parameter linear trend time series, may estimate the current value according to the historical ones, which could be helpful to weight estimation.

Assume the true value of the weight at time t is W_t. Besides, assume that the first and second exponential smoothing result is $S_t^{(1)}$ and $S_t^{(2)}$, the smoothing factor is a and the estimation cycle from time t is 1, then the estimation formula is shown below [12].

$$\begin{aligned} S_t^{(1)} &= aW_t + (1-a)S_{t-1}^{(1)} \\ S_t^{(2)} &= aS_t^{(1)} + (1-a)S_{t-1}^{(2)} \\ \hat{W}_{t+1} &= \frac{2-a}{1-a}S_t^{(1)} - \frac{1}{1-a}S_t^{(2)} \end{aligned} \quad (4)$$

III. Implementation of Improved Dynamic Load Balancing Algorithm

A. Framework of Improved Dynamic Algorithm

The improved dynamic load balancing algorithm we suggest is based on the WLC scheduling algorithm. Besides, *Keepalived* has been used to implement health checking and weight update of server nodes. To be specific, the MISC_CHECK module of *Keepalived* allows a user-defined script or executable program to run as the health checker [6]. The exit code of the script or program can be used to update the LVS scheduling table. If the exit code is 0, weight of a server node remains unchanged. Computing service of a server node is unavailable when the exit code is 1. In addition to the two cases, the weight of a server node will be set to 'exit code-2' when the range of exit code is 2-255 [6]. According to this idea, we can achieve our dynamic load balancing algorithm through MISC_CHECK module of *Keepalived*. Each modules of this dynamic algorithm is shown in Figure 2.

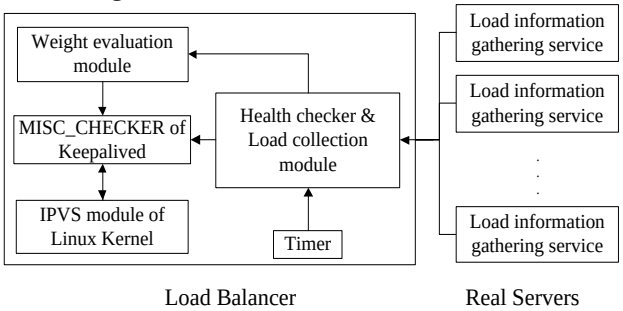

Figure 2. Modules of improved dynamic algorithm

The user defined module get the health status and the load information periodically. Then the weight evaluation module calculates new weight of the server node. The LVS scheduling table is refreshing through MISC_CHECKER module. If the health checking fails, the module will remove the server node from the server pool automatically. Else, the new weight of a server node will be update to the one calculated by weight evaluation module.

B. Flow of Improved Dynamic Algorithm

The load balancer collects load information periodically, so load gathering service should be running real-time. We collect load information above through some system files of Linux. The load balancer and the real server exchange load information by using the client and server communication mechanism, which are both user-defined. We set the program *mon_srv* running in the server node to gather and send load information to the load balancer. The program *mon_cli*, running on the load balancer, sends request to get load information periodically.

Computing service is also running real-time on the server nodes. The load balancer checks the health of a server node's computing service firstly by using *mon_cli*, and then gets the server response time if the server is health. Then the balancer gets load information from the load information gathering service. The load balancer checks the computing service by using TCP connection. If the service is healthy, the load balancer checks the scheduling table to check whether the node exists or not, and then get the response time. Else, the load balancer removes the node from the scheduling table and set the weight of the node to 0. After that, the load balancer collects load information from the server node by using UDP connection. Then the weight evaluation module calculates new weight of the server node by using our weight evaluation model. Finally, new weight of the server is updated by *Keepalived*. The flow chart of the algorithm is shown in Figure 3.

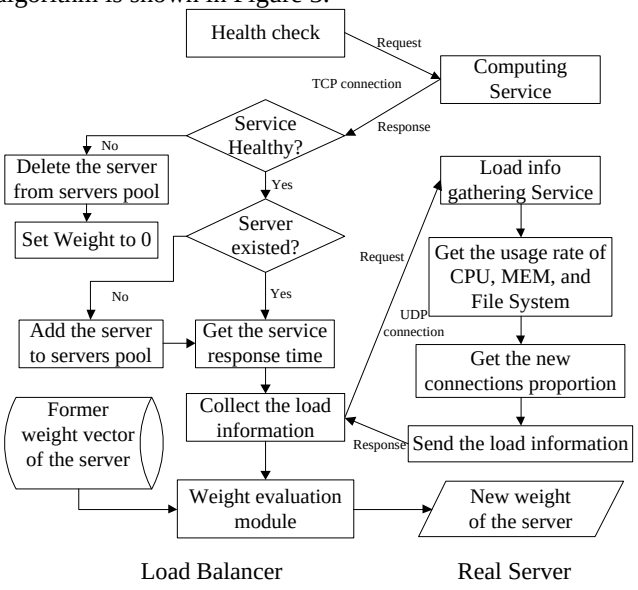

Figure 3. Flow chart of improved dynamic algorithm

IV. EXPERIMENTS

In order to test the performance of improved dynamic load balancing algorithm, we have set up an experiment environment, on which WLC algorithm, simple dynamic load balancing algorithm and our improved dynamic algorithm has been implemented.

A. Hardware Environment

In our experiments, 6 blade servers and 2 industrial computers has been used, among which 1 blade server serves as the client, 2 blade servers serve as load balancers, and the other 3 blade servers serve as server nodes together with the 2 industrial computer. The 6 blade servers use one switch, while the 8 devices use one. The hardware and OS parameters of the 8 devices are shown in Table I.

TABLE I. CONFIGURATION PARAMETERS OF HARDWARE

Parameters / Hosts	CPU (Core/GHz)	MEM (GB)	Storage (GB/R)	OS
Client	16/2.40	8	320/7200	Win 2008
Load Balancer(M)	16/2.40	8	320/7200	SUSE 11
Load Balancer(B)	8/2.40	8	320/7200	SUSE 11
Real Server 1	16/2.40	8	320/7200	SUSE 11
Real Server 2	8/2.40	8	320/7200	SUSE 11
Real Server 3	8/2.40	8	320/7200	SUSE 11
Real Server 4	4/2.26	2	160/5400	SUSE 11
Real Server 5	2/2.50	4	120/5400	SUSE 11

Hardware devices' configuration parameters are shown in Table 1. There are two load balancers to implement failover through VRRP, both of which have the same LVS configuration, virtual IP address, and *Keepalived* configuration [6]. Topological relations between the devices above are shown in Figure 4.

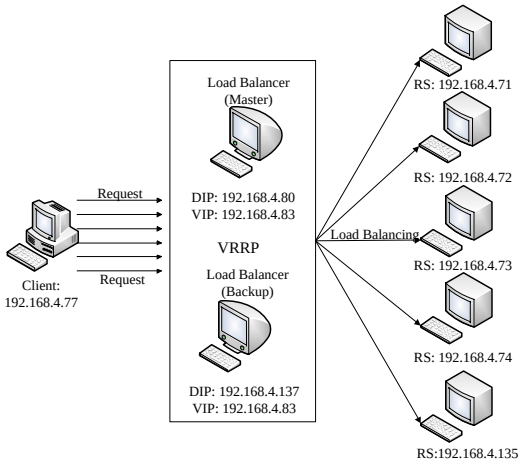

Figure 4. Topological relations between the devices

B. Software Environment

The operating systems of the devices have been shown in Table 1. Each server node supplies the same computing services and we choose five of them to do our experiments. The request processing time and the result data packet size of each service is shown in Table II.

TABLE II. EFFICIENCY AND PACKAGE SIZE OF COMPUTING SERVICES

Service Name	Request Processing Time (ms)	Size of Data Packet (KB)
te_tl_ksp_radix_multi	20	30
te_ts_ksp_radix_multi	10	20
te_ts_sp_bidij_buckets	40	35
te_ae_ksp_astar_heap_multi	60	40
price_svc	1	2

The '*price_svc*' service is the most efficient among these services, while the '*te_ae_ksp_astar_heap_multi*' service is the least. It's important to point out that, the two indexes are the average result of 430 test samples, and the latter one has more impact on the network load.

The computing services are compiled by GCC, while the client program is Java application, which can be generated to JAR file for testing. The open source software Apache JMeter has been used to simulate multiple clients, which could send requests to the server nodes. JMeter is an Apache top level project that can be used as a load testing tool for analyzing and measuring the performance of a variety of services. The concurrent test is implemented by using multi-threaded method [13]. Our experiments are performance test with Java application and the concurrent test and analyzing function of JMeter can meet our requirements.

A client and a server node communicate with each other through a TCP connection. The client should do login and authentication after a connection is established. Then the client send request data package with specific service name and parameters through the socket instance. After the login and authentication process, multiple requests can be send to the server node until the connection is closed. For each request, the service is chosen randomly in our experiments.

C. Content of Experiments

The test objects of our experiments is the original WLC scheduling algorithm, simple dynamic scheduling algorithm based on WLC, and improved dynamic scheduling algorithm based on WLC. The three algorithms can be recorded as WLC, DWLC and IDWLC for convenience. We use *JMeter* as the test and analysis tool and the test index is the throughput of the system shown in Figure 4.

The cycle index of the threads group in *JMeter* should be set to a constant. For each cycle, there are three parameters to adjust, which are number of concurrent connections, the ramp-up period of the concurrent threads, and number of requests per connection. We denote the three parameters as P, Q, and R. We study the system throughput variation tendency when parameters P, Q, and R changes, and then draw some conclusions of three algorithms through analysis.

D. Results and Analysis

In order to test the performance of three algorithms, we set the cycle index to 10 in each experiment, so as to make the test closer to the real situation. For the WLC algorithm, we set the weights of real server nodes in Figure 4 to 50, 50, 50, 40, and 40. For the DWLC algorithm, we set the original

 ISBN: 978-1-61208-187-8

weights of real server nodes the same to 50 and the expected value of the aggregate load to 0.70.

Then we determine the parameters of the IDWLC algorithm. Firstly, the ideal service response time and the true time should be determined through lots of experiments. As the average server response time is greater than 0.2 milliseconds in our 100000 experiments, we set the T_{ideal} to 0.2 milliseconds. Through the data analysis, we find that if the computing service is healthy, the range of response time will be 0.20-0.35 milliseconds. In order to set the parameter more properly, we set the upper bound of T_{delay} to 0.35 milliseconds. Then, we set the value of other important parameters. As the weight value range of LVS server node is [0, 253], we set w_0=20 for convenience. Considering the importance of former weights and load parameters, the vector **P** is set to [0.2, 0.3, 0.5], **Q** is set to [0.4, 0.2, 0.1, 0.3]. Through some experiments, we set expected value of the aggregate load A to 0.7, and initial weight W_0 to 50. According to the exponential smoothing prediction theory, the greater the fluctuation range of predicted target is, the more the predicted value depends on the true value of the previous moment [11]. As a result, the value of smooth factor 'a' ought to be greater. Considering the weight sequence variation, we set a=0.6. The weight prediction and adjustment formula of our experiments is

$$\begin{aligned} S_i^{(1)} &= 0.6W_i + 0.4S_{i-1}^{(1)} \\ S_i^{(2)} &= 0.6S_i^{(1)} + 0.4S_{i-1}^{(2)} \\ \hat{W}_{i+1} &= 3.5S_i^{(1)} - 2.5S_i^{(2)} \end{aligned} \tag{5}$$

$$W_{i+1} = \begin{cases} 50 + 10 \times \dfrac{0.2}{T_{delay}} \sqrt[3]{0.70 - \mathbf{QL}^{\mathrm{T}}} & i < 3 \\ \mathbf{PW}^{\mathrm{T}} + 10 \times \dfrac{\hat{W}_{i+1}}{\mathbf{PW}^{\mathrm{T}}} \left(\dfrac{0.2}{T_{delay}} \right)^{\mathrm{sgn}\left(0.70 - \mathbf{QL}^{\mathrm{T}}\right)} \sqrt[3]{0.70 - \mathbf{QL}^{\mathrm{T}}} & i \geq 3 \end{cases} \tag{6}$$

1) System throughput T and concurrent connections number P: We set the ramp-up period of the connections to 10 seconds, requests number per connection to 50. When concurrent connections number changes from 50 to 5000, throughput curves are shown in Figure 5.

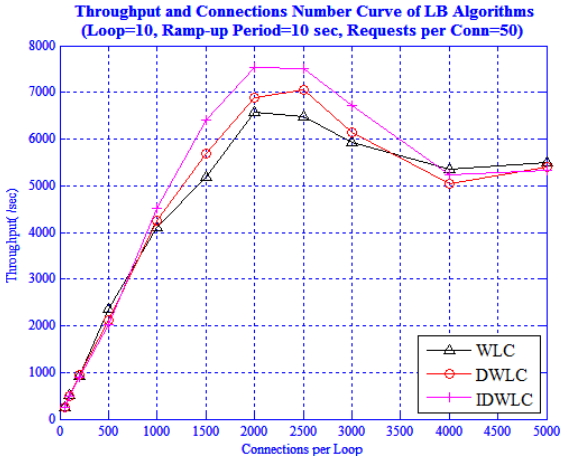

Figure 5. Throughput and connections number curves of three algorithms

As shown in Figure 5, when concurrent connections number is smaller than 1000, difference among throughputs of the system under three algorithms is insignificant. The reason is that, when the connections number is small, the processing ability of real server nodes is enough to handle the requests from the client, so the dynamic algorithms' balancing effect is no better than the WLC's. When concurrent connections number is greater than 1000, load of the server nodes increases, and the dynamic algorithms come into play and assigns the new connections more balanced. It is shown in Figure 5 that, when the connections number is greater than 1000 and smaller than 3500, the two dynamic algorithms achieve greater throughput than the WLC algorithm. Especially, our dynamic load balancing algorithm achieved greater throughput than the other two. When the number of connections is greater than 3500, the load of the server nodes is greater than the processing ability of them, and the effect of dynamic algorithm becomes insignificant compared with the WLC algorithm. As a result, system throughput of the three algorithms tends to be close to each other.

2) System throughput T and threads ramp-up period Q: We set the number of concurrent connections to 2500 and the number of requests per connection to 50. When the ramp-up period changes from 1 second to 40 seconds, throughput curves are shown in Figure 6.

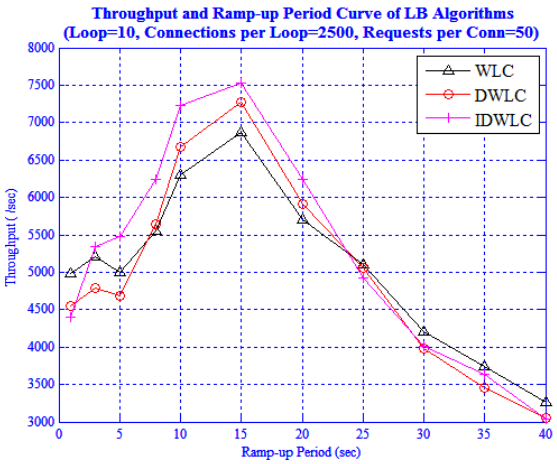

Figure 6. Throughput and ramp-up period curves of three algorithms

This experiment set the number of connections and number of requests constant, so the total load per cycle is constant too. As shown in Figure 6, when the ramp-up period is small, system throughput of three algorithms is low. The reason is that, the server nodes are under excessive load during the ramp-up period, which leads to queuing or waiting phenomenon and decreases the system throughput. When the ramp-up period increases, the queuing or waiting phenomenon has been alleviated. As a result, the two dynamic algorithms achieve greater throughput than the WLC algorithm and our improved dynamic algorithm shows better result than the simple one. When the ramp-up period is greater than 20 seconds, the assignment of new connections is sparse, and the load of the server nodes gets smaller,

 ISBN: 978-1-61208-187-8

which lead to almost the same throughput for the three algorithms.

3) System throughput T and requests number R per connection: We set the number of concurrent connection to 2500 and the ramp-up period to 10 seconds. When the requests number per connection changes from 10 to 160, throughput curves are shown in Figure 7.

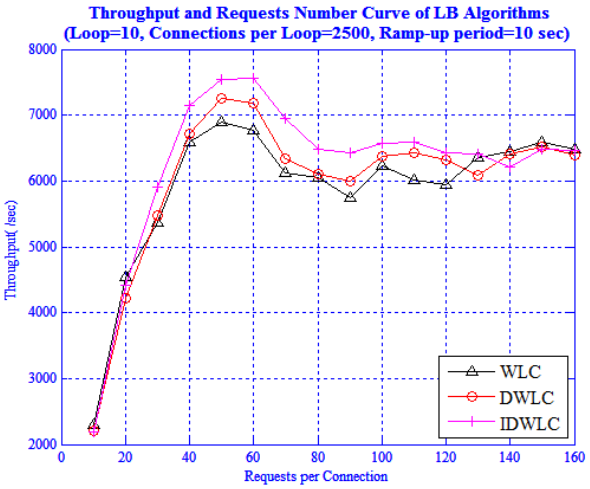

Figure 7. Throughput and requests number curves of three algorithms

This experiment set the connections number and ramp-up period to constant. When the requests per connection is smaller than 20, the load of server nodes is low. As a result, the effect of two dynamic algorithms is not better than the WLC algorithm. With the increase of the requests number, dynamic algorithms could assign new connections more proper, system throughput gets greater than the WLC algorithm. We can find in Figure 7 that, our improved algorithm achieves greater throughput than the other algorithms when the requests number changes from 30 to 120. As the requests number gets greater than 120, the total load is too heavy to the server nodes, which could lead to similar system throughput for the three algorithms.

V. CONCLUSION

As described above, a static load balancing algorithm is not sufficient to assign client connections when processing time requests vary, and thus the scheduling programs of the Linux Virtual Server are not useful [5]. A dynamic load balancing algorithm has proposed before to solve this problem. However, the algorithm has some problems, which may reduce its usefulness, and thus we propose a more efficient load balancing algorithm that achieves better results.

The improved model we suggest could solve the shortcoming of the simple dynamic algorithm and improve the stability of the dynamic scheduling process. For one thing, computing service response time has been used to adjust the weight variation, aims to highlight the important role of the network delay for load balancing. For another, the exponential smoothing estimation method has been used to make the adjustment of weight consistent with the actual variation of load. The experimental results show that, our improved dynamic load balancing algorithm could achieve greater system performance than the other two, if the total load is proper to the real server nodes.

ACKNOWLEDGMENT

This work was supported in part by the National Key Technology R&D Program under Grant 2009BAG12A08 and the Research Foundation of Easyway Company.

REFERENCES

[1] Linux Virtual Server Project, “Linux Virtual Server, ” 2007, http://www.linuxvirtualserver.org

[2] Zhang Wensong, “Linux Virtual Server for Scalable Network Services,” Ottawa Linux Symposium 2000, 2000(7)

[3] Suntae Hwang, Naksoo Jung, “Dynamic Scheduling of Web Server Cluster,” Proc. Ninth Int’l Conf. Parallel and Distributed Computer Systems (ICPADS ‘02), 2002

[4] Zhang Wensong, “Job Scheduling Algorithms in Linux Virtual Server,” 2005, http://www.linuxvirtualserver.org

[5] Zhang Wensong, “Dynamic Feedback Load Balancing Algorithm”, 2005, http://zh.linuxvirtualserver.org.

[6] Alexandre Cassen, “Keepalived for LVS datasheet”, 2002, http://www.keepalived.org/pdf/UserGuide.pdf

[7] Ma Wei, “A New Approach to Load Balancing Algorithm in LVS Cluster,” Master Degree thesis. Wuhan, Hubei, China: Hua Zhong Normal University. 2006.5.

[8] Shen Wei, “Research and Realization of an Improved Load Balancing Algorithm based on LVS Cluster,” Master Degree thesis. Bejing, China: China University of Geosciences. 2010.6.

[9] Qin Liu, Lan Julong. Design and Implementation of Dynamic Load Balancing in LVS. Computer Technology and Its Applications, 2007, 09:116-119.

[10] Yang Jianhua, Jin Di. A Method of Measuring the Performance of A Cluster-based Linux Web Server. Computer Development & Applications, 2006, 04: 58-60.

[11] Diao Mingbi, Theoretical Statistics. Beingjing: Publishing House of Electronics Industry. 1998.

[12] Luo Bin, Ye Shiwei, Server Performance Prediction using Recurrent Neural Network. Computer Engineering and Design, 2005,08: 2158-2160

[13] The Apache Jakarta Project, Apache JMeter, 1999-2011, http://jakarta.apache.org/jmeter

 ISBN: 978-1-61208-187-8