SLA-constrained Feedback-based Software Load Distribution Algorithm that Minimizes Computing Resource Requirement

S. R. Venkatramanan
PayPal
San Jose, CA
e-mail: raven@paypal.com

R. Hariharan and A. S. Murthy
eBay
San Jose, CA
e-mail: rehariharan@ebay.com and
asmurthy@ebay.com

***Abstract*—We describe a load distribution algorithm in this paper that uses the current transaction response time as feedback for dynamically routing traffic to a minimal number of machines that run a business function (pool), with the constraint to consistently meet the response time requirements. This enables us to dynamically vary the number of nodes as per traffic levels, traffic mixes, and varying node capacities - a typical scenario in cloud environments. First, we present details of the basic algorithm followed by an extended version. Both have been implemented and tested in the eBay private cloud. We include graphs that show how the number of active nodes vary with incoming traffic volume while preserving the response time requirements. Results of using the extended version illustrate how the performance of the mirror environment closely matches that of the real environment while running production traffic.**

Keywords-Software Load Balancer; feedback; SLA; load distribution; minimum nodes; energy optimization

I. INTRODUCTION

eBay's network sees varying amounts of traffic that show a diurnal variation, with traffic peaks during mid-day and evening hours. Early morning and midnight traffic varies from being half of the peak to even less, depending upon the business function served. Traffic handled by a typical node serving a typical business function on the network is shown in Fig 1.

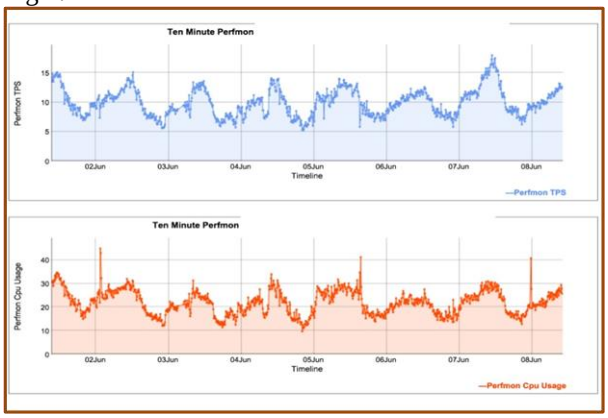

Figure 1. Traffic pattern seen in a typical node running a typical application

This means a number of nodes allocated to handle peak traffic of a given business function or application are idling for a significant part of the day. However, capacity managers are often uneasy about reducing the size of application pools during times of reduced need because any unexpected increase in traffic might result in the response time requirements not being met.

The above situation warrants two needs:

- An automatic way to detect the saturation state of nodes and augment the pool
- Ability to readily deploy and shutdown nodes as requirement dictates

Detection of the saturation state has to be dynamic and the system should be able to trigger the addition of new nodes in time, as needed, without affecting the application response time. It is possible to quickly add nodes capable of taking traffic in a responsive cloud setup using some of our earlier work, i.e., configure a node, deploy code, and bring it into traffic in a short amount of time.

In addition, if the right number of nodes in an application pool is dynamically managed, it can result in a significant reduction in energy consumption in the data center, in terms of power and cooling, while preserving the application response times through the course of the day, irrespective of traffic levels.

Nodes of the eBay cloud, about 185,000, come from various processor generations and systems technologies and thus, one Virtual Machine (VM) of a given size (CPU, memory, etc.) may vary significantly in capacity from another VM. Hence, using a round robin load dispatching will stress these nodes differently. In order to overcome the effect of varying technologies, weighted round robin is suggested. However, in a cloud environment, an application run by a guest VM does not have a dedicated environment on the host system and the background load on the host can vary considerably. This means the weights used would not only depend upon the technology used, but also vary with the other applications and their load on the host system. This performance variation is described in detail in [1]. Keeping the weights correctly defined becomes a complex task and does not guarantee the required transaction response time.

 ISBN: 978-1-61208-474-9

The main focus of this paper is to present a heuristic algorithm that minimizes the number of nodes needed to handle the traffic at any time with a constraint of preserving response time needs. This sets the stage to power machines up as needed and take down machines when not required for an extended period of time in the cloud environment.

The rest of the paper is organized as follows. In Section II, we present a summary of the types of routing algorithms used, stating how our algorithm differs from those algorithms. Section III has the description of the basic algorithm, where the requests and their response time distribution are more or less similar. Since the algorithm relies on routing requests to a minimal number of nodes, we also detail a heuristic method to route transient bursts in traffic in the case where the maximum number of nodes available is less than necessary for that traffic level (degraded operation). Section IV contains the changes needed to extend this procedure to heterogeneous traffic. We detail in Section V how we verified the performance of this algorithm. Section VI has the detailed results for both implementations and Section VII presents our conclusions.

II. CURRENT WORK IN CLOUD TRAFFIC ROUTING ALGORITHMS

Load balancing algorithms mainly fall into two categories- static algorithms and dynamic algorithms. According to recent survey papers [2][3], optimal routing algorithms to maximize throughput used in the cloud are based on shortest queueing with maximum weight scheduling at each server. The same policy is shown to be queue length optimal. These algorithms are also shown to be optimal for resource usage under heavy traffic conditions. Algorithms described in these surveys are all dynamic and are based on stochastic arrivals. In all the algorithms addressed in the literature, the number of servers traffic is routed to is given apriori. All optimization is done to either maximize throughput or to provide the best experience for the request, given the set of available servers.

Another survey paper by Katyal et al. [4] presents a thorough classification of routing requirements in the cloud and various algorithms that cater to these requirements. Static algorithms are based on routing to a given set of IP addresses or given machines that have the needed resources. Dynamic algorithms, presented in this paper, use the state of the system to route the request

There is a detailed comparison of various types of load balancing methods presented in a recent paper [5] that proposes the development of a new method for getting improved response times from servers. Active VM Load Balancer [5] comes close to what we propose, among the methods described in that paper. It takes into consideration the number of requests currently allocated to a server in deciding where to route the next incoming request. However, all available nodes are always open to receiving traffic and traffic is routed to nodes such that the quality of service is best. None of the algorithms described in [7] (Round Robin, Weighted Round Robin, Throttled Load Balancing, Dynamic Load Balancing using system utilization, and Active VM load balancer) vary the number of nodes to which traffic is routed.

None of these algorithms surveyed deal with routing to a minimal number of servers at any time. What makes our work unique is the fact that our algorithm routes the requests to a minimal number of nodes, freeing up unused nodes while maintaining the application response time requirement.

III. BASIC ALGORITHM DESCRIPTION

All commands or requests (embedded in the URL) come to the Software Load Balancer (SLB) and are forwarded to the node of choice. In the basic version of the algorithm, we only consider requests corresponding to similar response times. Subsequently, we extend this to more realistic environments with heterogeneous requests.

A. Algorithm Principle

Little's Law [6] describing the relationship between response time and number of requests in the system states

$$L = \lambda \times W \quad (1)$$

where, 'L' is the number of requests in the system, λ is the arrival rate of requests, and 'W' is the expected response time. In other words, response time 'W' and the number of requests in the system 'L' are linearly related. So, by controlling the maximum number of requests (or *connections* to a node, and equivalent to L in the above formula), we can control the response time effectively. This is the main idea behind this algorithm.

In the eBay private cloud, Service Level Agreement (SLA) for various commands is typically governed by the distribution of the response times. Henceforth, we will refer to the response times as SLA in this paper. We use the *historical median* of the currently observed response time, with consideration to the time of day, for baseline (*SLA-med*), and the *2nd standard deviation* (*SLA-95*) for tolerance. We accumulate the deviation of observed transaction response time of a node from *SLA-med* for each of a predefined set of transactions. As the accumulated value exceeds a certain predefined threshold proportional to *SLA-95*, we adjust the value for the maximum allowed *connections* into that node. To illustrate this, consider the following example. Let the median response time for a given command equal 100ms (*SLA-med*=100ms) and the 95th percentile (~2nd

standard deviation or P95) equal 200ms (SLA-95=200ms). Now, if the next series of response times are 120ms, 90ms, and 150ms, then the differences accumulated would be 20ms, -10ms, and 50ms respectively, summing up to 60ms. When the sum of response times of transactions up to a given number exceeds a given threshold, we take action by reducing the maximum number of *connections* allocated to that node.

To understand the statistical basis of this, let each response time x_i be normally distributed with a mean μ and standard deviation σ. If $y_i = x_i - \mu$, then the sum of k such differences, $\Sigma_i\ y_i \sim N(0, \sqrt{k} * \sigma)$. So the difference will exceed $+/-3*\sqrt{k} * \sigma$, with a probability < 1%. When this happens, this either signifies a very rare occurrence or it shows a shift in distribution of the variable, implying that the response times are either systematically increasing or decreasing. If the response times are increasing, we could control it by decreasing the maximum connections we allow into the node and vice-versa. For normal distribution, mean should be equal to median; however, our empirical distribution is not normal and has outliers. Hence, in our method we choose median instead of mean as it is more stable and not influenced by outliers. Further, since the traffic volume and response times, influenced by the traffic, are not temporally constant, median is calculated over a moving window and SLA, taken from the historical data, is changed over time.

The Software Load Balancer (SLB) maintains an ordered list of nodes and attempts to send the requests to nodes in that order. If a node that occurs earlier in the ordered list has *connections* available, then the request is sent to that node. The requests will use a minimal number of nodes as a result of maintaining an ordered list and sending traffic to the earlier nodes, until they cannot serve the request within the required service time constraint, likely because of lack of resources on that node. We can tune this threshold to achieve the necessary SLA by controlling the maximum *connections* into a node. Note that the action to increase or decrease the *connections* is based only on the SLA target (median and P95 values), which are essentially surrogates used to represent resource utilization. It is to be noted that poor performing applications will also be exposed by this.

B. Procedure

Flow charts in Fig. 2 and Fig. 3 show how the load distribution algorithm works in the SLB. Fig. 2 shows how the maximum number of *connections* (*CMax*) is dynamically adjusted for any given node after each transaction (*N*) is completed by that specific node in the pool. Each node is initialized with its own *CMax* and they are based on historical observation of traffic handling capability of the overall pool, taking into consideration the time of day, the day of week, and the season. Its value is adjusted, as described in the flow chart, based on the value of an accumulator which essentially maintains the sum of differences between the actual observed response times seen at the node and the expected response time or SLA.

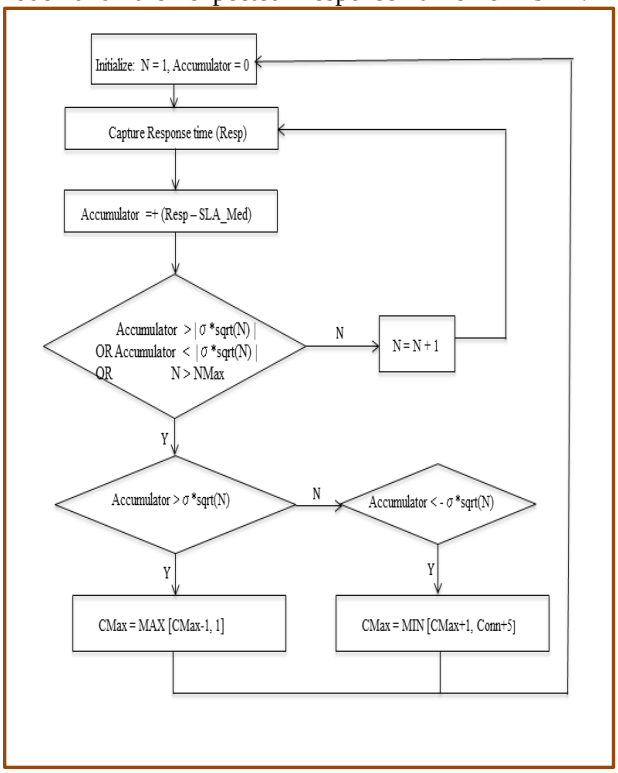

Figure 2. Basic Flow Chart to determine Max connection for each node

The accumulator is reset whenever either a predefined maximum number of transactions (*NMax*) have been completed by the node or when *CMax* is modified as a result of the accumulator exceeding certain thresholds. A buffer of 5 connection counts while decreasing, prevents the reduction of connections too soon and from being unable to accommodate any surge in traffic while idling. It also ensures traffic is not black-holed into one node when transactions complete too fast because of error returns or bugs in the application.

C. Node choice and Degraded Operation

Fig. 3 shows how a choice of the node to route the request to is made when a new request comes into the SLB. It is to be noted that when there is no node found fit to route the request to (a case when there should have been more nodes made available but not fulfilled for whatever reason), the load distribution procedure drops to a lower grade of service level. Degraded service level is applicable only in cases when the system does not provision additional nodes in time.

Routing can use the following in such cases.

- Simple Round Robin or random routing to the available nodes.

- Use of a relaxed SLA (gradually increasing the SLA by 10% at a time) to determine a new *CMax* for each node and routing the request to the first node with available connections. We will refer to this method as Relaxed SLA.

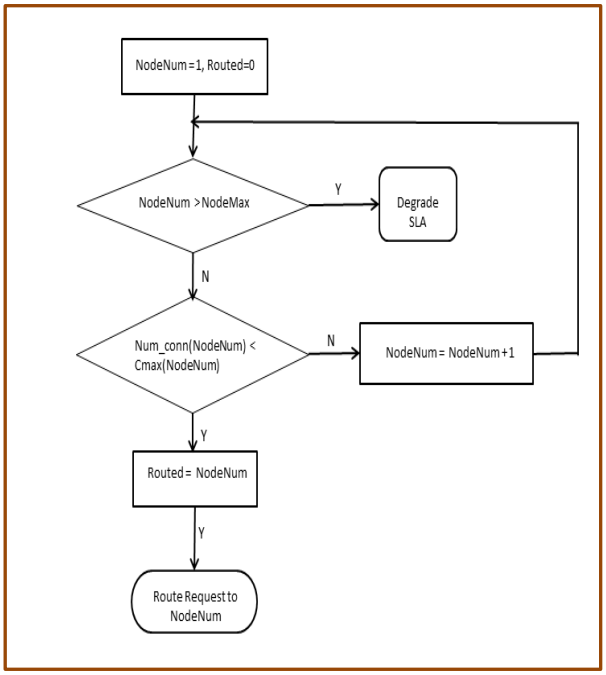

Figure 3. Basic Decision Flow Chart

Details of degraded operation will be beyond the scope of this paper though both of the following methods have been implemented and verified. We use the latter method, in the final implementation, by always maintaining a secondary *CMax* value corresponding to relaxed SLAs. In a situation where no node is considered available under the relaxed SLA, we repeatedly try from the first node, using the secondary *CMax* (that is 10% higher than the value at the previous level of relaxation) until a node, if any, that permits traffic to be routed is found. This process of relaxing SLA is done iteratively.

IV. HETEROGENOUS ENVIRONMENT

The preliminary version of the algorithm described above is only applicable to a homogeneous environment where all requests have a response time requirement of the same order. However, a single eBay application can handle requests of different types with varying response time needs, ranging from a few milliseconds to nearly a second. If incoming requests to such an application is handled as a single type with a large variation, this large variance makes it difficult to effectively adjust the *CMax* value.

We extend the basic version of the algorithm by grouping commands with similar response times. Using the historical median and standard deviation of the response time for a command as input variables, we use *KMeans* to classify the commands into a limited number of groups. The number of groups is determined by the number of peaks observed in the distribution of response times of all commands. For a distribution as shown in Fig. 4, the number of groups will equal to 2. In cases where multiple modes in the distribution come from the same command, due to multi-modal distribution of the response times of the same command, the number of groups to classify the commands should be appropriately reduced. Once the number of groups is chosen, *KMeans* classification is used to group the commands.

In a production environment, the behavior of commands and their respective response time distribution varies continuously, mainly due to variability in user behavior. Therefore, the grouping process is repeated each hour to ensure minimum variability within a group.

Once the commands are classified into multiple groups, we introduce SLB Group Modules as in Fig. 6, with each module handling only commands for a single group.

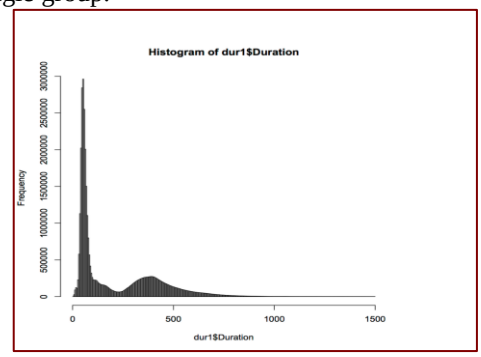

Figure 4 - Histogram of response distribution of all commands

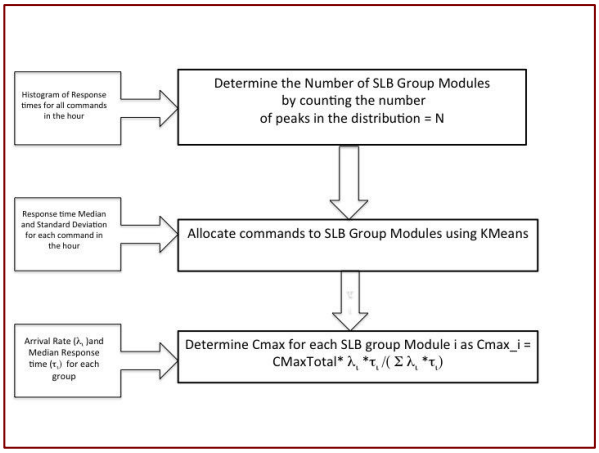

Figure 5: Allocation of Commands to groups and Seeding CMax for each group

The next step is to allocate a base number of connections for each group. This is done by weighting the total number of connections according to arrival rate * median response time for each group. Fig 5 summarizes this process.

Fig 6 shows how commands are routed to individual SLB Group Modules by the master router.

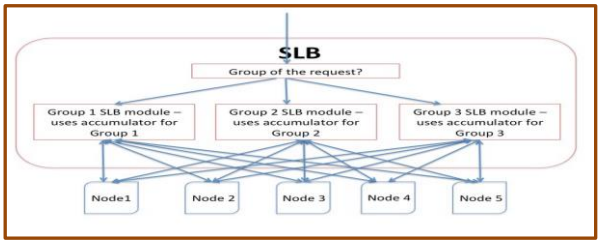

Figure 6. Routing of commands to SLB Modules in a Heterogeneous environment

V. VERIFICATION ENVIRONMENT

Algorithm verification was accomplished as follows. A typical eBay application (in Java) was deployed on 4 nodes each configured with 4 processors and adequate memory, and backed by necessary services and databases. Transactions executed by this application during a typical day were captured from the production application logs and they were grouped by similar measured response times. This provided a workload to be later played back by JMeter [7] instances and targeted to the SLB running various algorithms. In the first phase, we restrict the transactions to a single group with homogeneous response times; the later phase will include transactions with wider response times.

Multiple Jmeter instances were used to control traffic rate and patterns. Performance metrics were obtained through JMX interface built into the application as part of routine measurement infrastructure. This infrastructure provides measurements such as throughput (Transactions Per Second), CPU utilization, and Transaction Response Time, besides JVM heap related metrics, aggregated over a selected interval such as 1 minute, 10 minutes, or one hour. For short experiments of an hour or two durations, 1-minute aggregation is used. In the first set of experiments presented, we use round-robin routing to handle the degraded state of operation.

VI. RESULTS

Here we discuss the results of executing this algorithm showing the traffic arrival pattern as well as the corresponding performance trend of each of the nodes with respect to the elapsed time. Fig. 7 shows the intensity of the load in terms of active users on the system as time goes by. Fig. 8 shows the throughput achieved by each of the nodes, their corresponding CPU utilizations, and response time of the requests on corresponding nodes. Discontinuities in the later part of Fig 8 and Fig. 9 are because of the measurement infrastructure dropping measurement data.

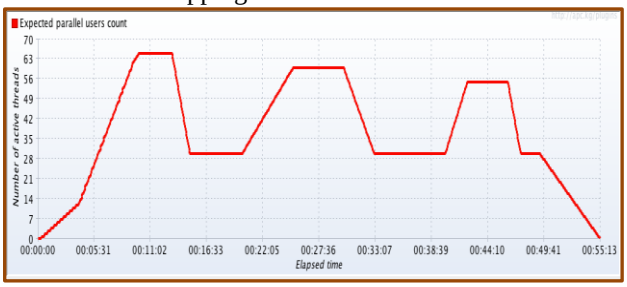

Figure 7. Arrival Pattern with multiple load variations

Fig. 8 demonstrates the recoverability of the system following this algorithm in a cyclical traffic pattern, including a small burst (from 9 to 12 minutes into the test) that was not compensated by an increase in the number of nodes resulting in a brief loss of response time SLA (800ms). However, as the load eases up after about 13 minutes into the run, CPU utilization starts to come down, also bringing the response time under SLA. Eventually, when the traffic slows down considerably, nodes start to drop off in LIFO fashion. As can be seen from the CPU utilization plot, Fig. 8b and Fig. 9b, this algorithm provides for completely removing a node from the pool between 15 and 20 minutes and, again, between 32 and 45 minutes of this abridged test.

The reader is encouraged to also note that one of the nodes shows 25% CPU utilization even before traffic begins in the graphs in Fig. 8b and Fig. 9b. Driving one of its 4 CPUs into an artificial loop and consuming 25% of the available resources purposefully degrades that node. This is to demonstrate the resilience of the algorithm when nodes of heterogeneous capabilities are presented, or during misbehavior of any of the nodes in the pool that may be unavoidable but not easy to extricate in time.

 ISBN: 978-1-61208-474-9

TABLE I. COMPARISON OF RESPONSE TIMES

	Median			P95			Count
Command	Target	Ref	T-R in %	Target	Ref	T-R in %	Ref
AdvS	123	109	12.84	262	238	10.08	3892
AllD	103	112	-8.04	208	218	-4.61	5540
ChsM	457	459	-0.44	1228	1028	19.44	8960
Cust	43	39	10.26	138	138	0.00	1206
FavS	19	222	-91.44	106	417	-74.47	1010
FndH	637	613	3.92	2236	2881	-22.37	43
FndM	90	96	-6.25	1098	5315	-79.34	593
GetC	76	75	1.33	76	75	1.33	1
JsDi	569	559	1.79	1054	1066	-1.11	11625
Prev	163	160	1.88	260	242	7.44	722
RecC	363	368	-1.36	517	544	-4.96	82951
SvSD	320	322	-0.62	549	558	-1.61	273873
SePr	477	477	0.00	682	686	-0.45	1631
SRPR	679	676	0.44	1400	1432	-2.23	1444307
SRSS	560	561	-0.18	1111	1170	-5.04	200592
SelO	585	579	1.04	1060	1059	0.09	62547
SimI	575	583	-1.37	915	985	-7.11	131498
V4Aj	244	104	134.62	323	196	64.96	4
Vero	832	750.5	10.86	2830	907	212.14	2
ZipP	44	42	4.76	96	71	35.21	426
TOTAL							2231423

Table I summarizes a test run where live traffic from multiple servers was mirrored into the SLB created to handle heterogeneous requests with 3 groups. The servers allocated to the SLB were part of the eBay cloud, thus representing the same environment as the current servers. The reader is drawn to the highlighted row that shows how response time requirement was met, both median as well as 95th percentile, for the most predominant type of request in a typical traffic composition of the peak day of the week by executing the algorithm.

Resource Consumption Ceiling

Brief loss of SLA, from 9 to 12 minutes into the test, (Fig. 6c) was mainly because CPU was driven to about 95% utilization leaving insufficient processing capacity even for basic bookkeeping functions of the system. Adherence to the set response time SLA can be controlled by introducing an additional constraint on maximum resource utilization, or running the application at a slightly lower priority that gives system processes an opportunity to do their functions necessary for the stability of the system.

An experiment was conducted to simulate this constraint by limiting the incoming traffic to use just under 90% CPU and the results given below in Fig. 9b are indeed encouraging in confirming that observation.

VII. CONCLUSION

We have presented in this paper an algorithm for efficient allocation of resources while adhering to response time requirements of applications under varying load conditions in cloud environments. The full potential of this algorithm can be realized with an adjunct system to flex up and flex down the nodes as needed.

Future work should include a mechanism for a trigger to add and remove nodes, and has built-in hysteresis to avoid frequent add/remove. The trigger mechanism should have complete knowledge of the cloud environment and should provide enough lead time based on provisioning time needed by the underlying cloud management system and the traffic intensity or rate of change in the traffic.

ACKNOWLEDGMENT

Authors would like to acknowledge Rami El-Charif, former Technical Fellow at eBay Inc., for his valuable contribution in discussions and guidance throughout this exploratory design and implementation of this algorithm and eBay management for its support.

REFERENCES

[1] Hariharan R, Murthy A.S., and Venkatramanan S. R., “How to handle noisy neighbors?”, CMG Conference Proceedings, 2014, pp. 345-351.

[2] Kuppuswamy, K, and Mahalakshmi, J, “A survey on routing algorithms for cloud computing, IJCA Proceedings on International Conference on Computing and information Technology 2013 IC2IT (4), pp. 5-8, December 2013.

[3] Mohana, S. J, Saroja, M, and Venkatachalam, M, “Cloud balancing- A survey”, International Journal of Engineering Research and Development, Volume 8, Issue 8 (September 2013), pp. 13-17

[4] Katyal, M, and Mishra, A, “A comparative study of static and dynamic load balancing algorithms”, International Journal of Distributed and Cloud Computing, Volume 1 Issue 2 December 2013, pp. 5-14

[5] Zaouch, A. and Benabbou, F, “Load balancing for improved quality of service in a cloud”, International Journal of Advanced Computer Science and Applications, Vol 6, No. 7 (2015), pp. 184-189.

[6] Little, J. D. C. (1961). "A proof for the queuing formula: L = λW", Operations Research 9 (3), pp. 383–387. JSTOR 167570

[7] Apache Foundation. “JMeter: Graphical server performance testing tool” Available for download at http://jmeter.apache.org/download_jmeter.cgi, Last Access Date: 21APR2016

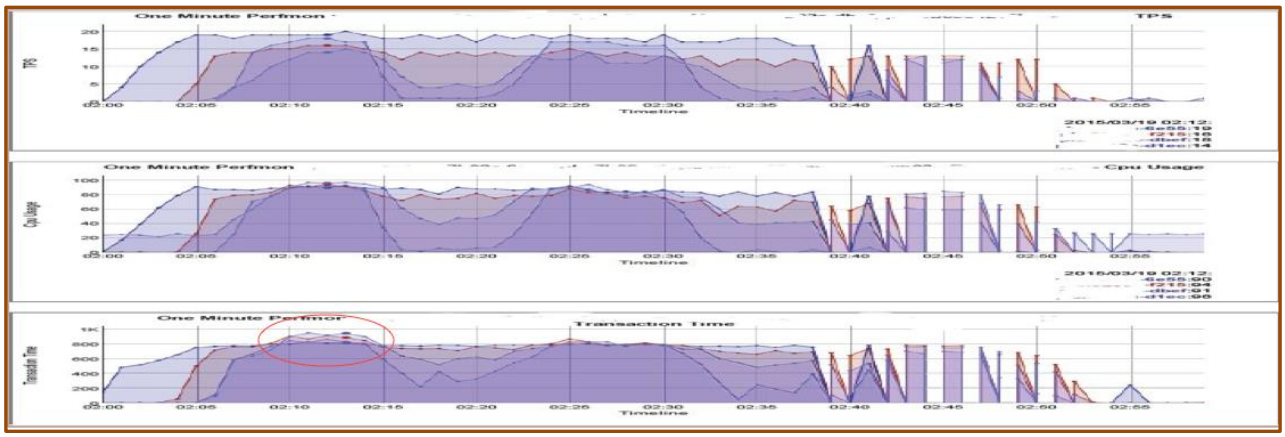

Figure 8. Under-provisioned System - SLA Violation at peak load. 8a – TPS, 8b – CPU, 8c – Transaction Time

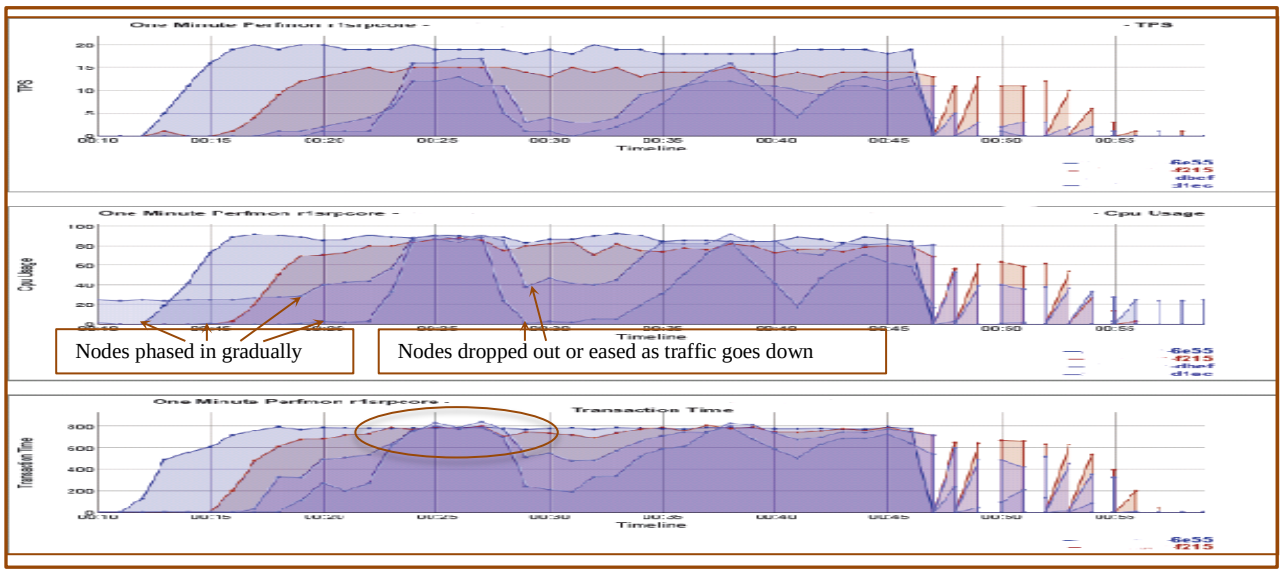

Figure 9. System Behavior under controlled load – Opportunity to remove a node under light load. 9a – TPS, 9b – CPU, 9c – Transaction Time