A Control-based Load Balancing Algorithm with Flow Control for Dynamic and Heterogeneous Servers

Rodolpho G. de Siqueira[1], **Daniel R. Figueiredo**[1]

[1]Programa de Engenharia de Sistemas e Computação
Universidade Federal do Rio de Janeiro (UFRJ)
Caixa postal 68.511 – 21.941-972 – Rio de Janeiro – RJ – Brazil

Abstract. *Although load balancing is a fundamental and well-studied problem in resource allocation, the ever changing scenarios and technologies in distributed systems demand new approaches and algorithms. In this context, we consider a real world scenario where servers are heterogeneous and have dynamic background loads not controlled by the load balancer. In such scenarios, classic round robin policy or novel joint the shortest queue policy are not effective. We propose a load balancing algorithm that dispatches requests to a set of heterogeneous servers according to their CPU availability using a feedback control loop to prevent overloading. We implement this policy and evaluate its performance in real and controlled scenarios. Our evaluation indicates the proposed algorithm is more effective in distributing load than other classic policies, in particular when background load is dynamic.*

1. Introduction

Load balancing is a fundamental block in the design of many distributed systems since it provides for sharing demand (i.e., load) across a set of resources. Not surprisingly, load balancing mechanisms have been broadly studied from a theoretical perspective and is widely applied to real systems, most recently in the context of data centers and cloud services [Patel et al. 2013].

The tradeoffs and effectiveness of load balancing mechanisms greatly depend on the scenario under consideration. For example, load balancing web traffic in a data center is very different from load balancing backbone traffic in an ISP (Internet Service Provider). Underlying assumptions on the available information, such as the state of resources (e.g., current CPU load) and the demand (e.g., processing time of a request), as well as assumptions concerning dynamic aspects of resources and demand, greatly influence the design of an effective load balancing mechanism. Indeed, despite its long history, research and development on load balancing mechanisms continues [Gandhi et al. 2015], as different scenarios and assumptions emerge from technological advancements and new applications.

A common assumption in the design of load balancing mechanisms is that servers receive requests (i.e., load) only from the load balancer (one or more). Another common assumption is that the load balancer is rather "close" to the resources, in the sense that processing a request takes longer than sending the request to the resource. In this paper, we consider a real world scenario where these and other common assumptions do not hold, prompting the design of a new load balancing mechanism.

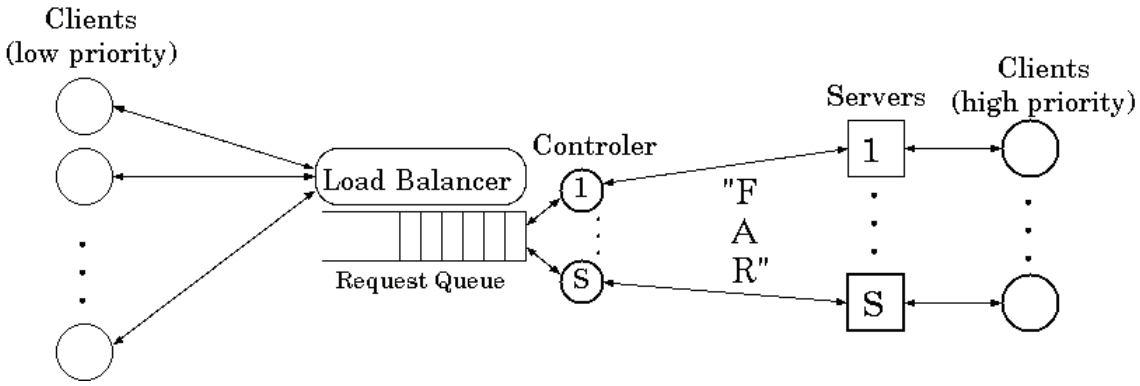

Figure 1. Illustration of the scenario considered where servers receive high priority demands from nearby clients while the load balancer dispatches low priority requests from far away.

In particular, we consider a scenario where a large set of clients need to monitor real-time variables stored on a small set of co-located servers. These real-time variables are replicated across the servers on local databases. However, the servers run mission critical applications that have high priority and unpredictable behavior. Thus, requests for reading real-time variables have low priority and must not impose a load that interfere with dynamic high priority demands. Last, servers have heterogeneous resources (e.g., different CPUs) and low priority clients are "far" from the set of servers (e.g, RTT is longer than the time to read a real-time variable). Figure 1 illustrates the scenario. This scenario is commonly found in oil platforms where inland clients need to monitor real-time variables in offshore oil platforms that run legacy servers and mission critical applications.

In this paper, we propose an effective load balancing mechanism for the above scenario. In particular, the load balancer dispatches aggregated requests to the servers according to a feedback control mechanism. The request rate sent to a server is determined by a corresponding feedback control mechanism that reacts to the CPU utilization on the server, an information periodically probed by the load balancer. We design the feedback control mechanism using a simple model for the relationship between number of requests and CPU load.

We implement the proposed mechanism and evaluate its performance under different scenarios, also comparing with the traditional Round Robing load balancing mechanism. Our results indicate that the proposed mechanism is effective in using the available resources in the servers without overshooting the pre-specified target CPU utilization. Moreover, the mechanism adapts to changes in CPU utilization of the servers due to high priority demands, moving low priority requests to under utilized servers.

The remainder of this paper is organized as follows. Section 2 poses the problem and the challenges in devising a load balancing mechanism in this context. Section 3 describes the feedback control for the proposed load balancing mechanism. Section 4 presents the evaluation of a real implementation of the proposed mechanism when running in different scenarios. Section 5 reviews the relevant work related to load balancing and control algorithms within the context of Dynamic Resource Provisioning. Finally, Section 6 exposes our conclusions and possible future work.

2. Problem formulation

As discussed in Section 1, we consider a real world scenario in which a small set of servers must respond with high priority to mission critical applications whose behavior are not easy to predict, but that remain idle for long periods of time. Thus, servers are mostly underutilized and their resources could be used more effectively, in particular when high priority applications are idle.

Consider the problem of remotely monitoring a very large set of real-time variables stored across this set of servers. Monitoring has low priority with respect to other applications running on these servers. Thus, resource usage by the monitoring application is restricted to a maximum budget, because servers must always be available to process high priority applications when needed. Moreover, since monitored variables are replicated across the servers, requests can be distributed across the servers, in particular proportionally to their CPU availability (and below the target) to minimize the chances of overloading the CPU when high priority applications suddenly run in any given server.

In summary, three basic goals must be achieved, which we discuss below:

1. Distribute low priority load to replicated but heterogeneous servers subject to uncontrolled high priority load.
2. Distribute low priority load to servers with CPU utilization below a pre-specified target.
3. Balance the low priority load across the servers in proportion to their CPU availability.

2.1. Load balancing

A common solution to load balancing is to adopt simple mechanisms such as Round Robin, and dispatch requests to servers accordingly. Nonetheless, since in our scenario the servers receive high priority demands from applications not subject to the load balancer, their load is not necessarily distributed proportionally to the available resources (i.e., CPU).

Other approaches use feedback signals from the servers to dispatch requests - for example, the balancer sends the request to the server with most available resources. When not even the most available server should receive further requests because it is operating with a CPU utilization above the pre-specified target, the balancer could simply refrain from sending requests. However, this kind of policy may lead the request rate to oscillate between two extremes, full-rate or no flow, which introduces jitter. As our client applications are for monitoring purposes, we propose a solution which leads to a steady, controlled flow instead. This results in less oscillations and monitoring samples taken at a more regular period.

2.2. Flow Control

A possible solution to the problem of avoiding overloading is to always dispatch low priority requests to servers and make them explicitly reject this requests when overloaded. However, such alternative can place an extra burden on an already loaded server, as rejecting small requests may cost almost the same as processing them. Also, this mechanism wastes a lot of time sending the request and waiting for the rejection, since in our scenario the servers are located far from the clients.

An alternative solution to avoid overloading the servers is to implement a throttling mechanism on the client side, making clients perceive timeouts as a sign of server overload and then slow down the request rate. However, this solution requires changing the code running on the client application. Thus, this alternative is not transparent to the clients and makes existing client implementations incompatible with the system (all clients must be changed).

Thus, we propose a joint solution which mixes load balancing with overload avoidance. Our solution meets the pre-specified target CPU utilization of the servers while simultaneously balancing the load proportionally to their CPU availability. Moreover, since the control intelligence is in the load balancer, our solution requires no change to the clients or the servers.

3. Proposed mechanism

We design a load balancer which does not simply distribute load equally among servers. Instead, it dispatches requests according to the servers' CPU availability, measured by an out-of-band feedback channel.

The balancer has a single request queue, from which requests are bundled and sent to the servers (see Figure 1). A maximum number of requests per second is removed from the queue and sent to a given server, as defined by the controller. The control is not performed jointly, but occurs independently for each server, following the same control algorithm. So, if two servers have the same amount of available resources, they will receive the same request rate; if one has more resources than the others, its controller will send it more requests; if a server's CPU utilization is above some target level, its request rate becomes negligible.

3.1. Controller

We propose a controller design which shall be applied independently to every server. Thus, we note that every parameter mentioned applies to a single controller instance, designed for a specific server.

In order to control the CPU utilization, we measure it at a constant frequency, taking samples every t_s seconds, which is the minimum time interval necessary for the measured value to change. If the controller takes decisions within a time interval smaller than t_s, then it will see the same previously measured value and may prematurely change its decision. Thus, we design a discrete-time controller which updates its control decision variable every t_s seconds, after having a new feedback measurement.

The balancer sends requests to a server as bundles. When it completes sending a bundle, it waits t_d seconds and then sends another one, which contains N_t requests to that given server. The value of N_t is decided for each server by the controller every t_s. Since the request rate to the servers at time instant t depends on the ratio between the number of requests N_t sent to the server and the interval t_d, we could control the rate by varying either of those values. As the timing requirements of the monitoring application for which the controller is intended are not very strict, we make t_d constant and vary N to achieve the request rate control.

In order to design the controller, we first determine a simple model of the steady-state relationship between CPU utilization of an idle server and the number of request

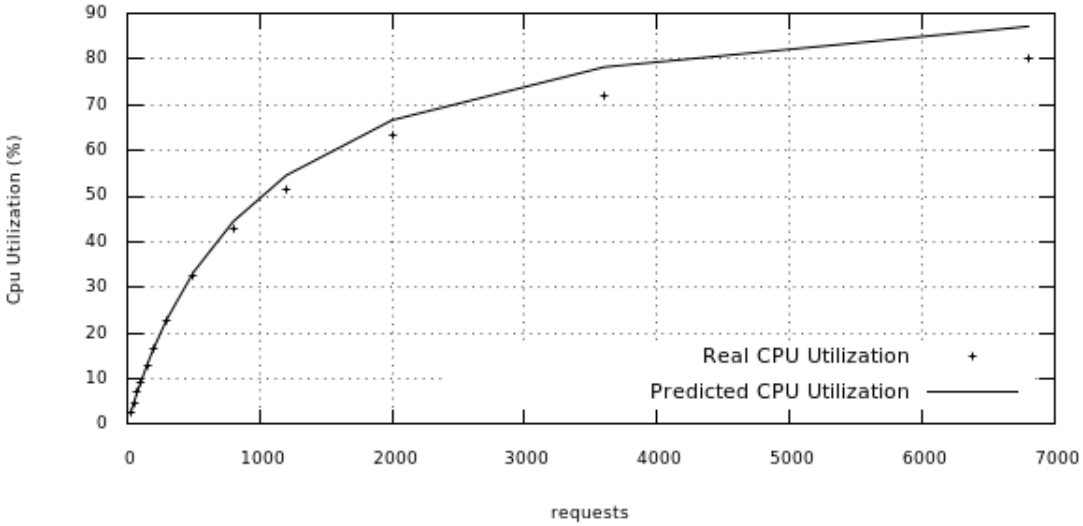

Figure 2. Comparison between model prediction and real system behavior (CPU utilization as a function of number of requests sent periodically).

received and processed every t_d seconds. This simple model assumes the CPU is idle - which is the case most of the time within our assumptions, as the high priority applications only runs sporadically.

Using this assumption, we estimate the CPU utilization through the ratio between the time spent processing a bundle of requests and the total time between the receiving of two consecutive bundle of requests. Let C_t denote CPU utilization at time t, N_t the number of requests in a bundle at time t, t_p the time required to process a single request, and t_d the period of the balancer (which sends a bundle of requests every t_d time units). Thus, we have the following ratio:

$$C_t = \frac{N_t t_p}{t_d + N_t t_p}, \tag{1}$$

where we have assumed that the network delay is negligible when compared to t_d and to the time required to process the bundle of requests, $N_t t_p$.

Note that this is a steady-state model while in practice there is a one-minute moving average dynamic relationship between the number of requests and the measured CPU utilization. We consider this dynamic relationship when operating the controller. Note also that t_p may be different across a set of heterogeneous servers, and hence the model can be adjusted for each case. Figure 2 shows the predicted value and the actual measured value of CPU utilization (in steady-state) as a function of the number of requests sent to the server, indicating a very good agreement, in particular when the number of requests is small. As the number of requests increases, the model assumption starts to be violated as the time required to transmit the requests become less negligible. Nonetheless, the model prediction is still in good agreement, and this simplified model has proved to be a good enough approximation for our purposes.

With the model relating C_t to N_t, we can evaluate how much N should vary in order for C to vary a desired amount. In particular, assuming a first-order approximation

of C as a function of N, we have the following:

$$\begin{aligned} C_t - C_{t-1} &= (N_t - N_{t-1})\frac{dC_{t-1}}{dN_{t-1}} \\ &= (N_t - N_{t-1})\frac{d}{dN_{t-1}}\left(\frac{t_p N_{t-1}}{t_d + t_p N_{t-1}}\right) \\ &= (N_t - N_{t-1})\frac{t_p t_d}{(t_d + t_p N_{t-1})^2} \end{aligned} \tag{2}$$

Considering that C is measured by a moving average, we cannot expect to control C in less time than the window over which the average is calculated, t_w. Hence, the controller should try to make C_t approach C^*, the CPU utilization set-point, by varying N through a process which takes place in an interval proportional to the time window t_w - the controller should take longer to achieve the reference if t_w is long, and make smaller variations to N. Note also that, since the controller only updates its control decision when it receives a feedback sample, once every t_s seconds, the variation in the number of requests should be proportional to t_s - if samples are not taken frequently, the measured CPU utilization shall have varied more from one sample to another, and then the controller needs to be more intense each time it has a chance to act.

With that reasoning, we come to the conclusion that C_t should approach C^* by a fraction $\frac{t_s}{t_w}$ of the difference $C^* - C_{t-1}$ every t_s seconds:

$$C_t - C_{t-1} = \frac{t_s}{t_w}(C^* - C_{t-1}) = (N_t - N_{t-1})\frac{t_p t_d}{(t_d + t_p N_{t-1})^2} \tag{3}$$

This yields a non-linear integral controller, which, at each instant t, increments the allowed request rate to a server based on a fraction of the difference between the reference CPU utilization, C^*, and its current measured value:

$$N_t = N_{t-1} + \frac{t_s}{t_w}\frac{(t_d + t_p N_{t-1})^2}{t_p t_d}(C^* - C_{t-1}) \tag{4}$$

4. Evaluation

We implemented the proposed load balancing mechanism and evaluated its performance in a controlled environment consisting of four Linux machines: a client, the load balancer, and two servers. We report on actual resource usage as reported by Zabbix, a common Network Monitoring tool. In particular, the load balancer machine runs a Zabbix poller, and the other machines run Zabbix agents that report the values for CPU utilization and network bandwidth.

We also evaluate and compare with the performance of a slightly modified round robin load balancer. This modifications sets a limit to the maximum request rate sent to each server in order to respect the target CPU utilization - using equation 1. However, this limit is computed assuming the servers are idle and is fixed - no adjustments to the maximum rate is performed during the experiments. Note that this modification avoids overloading any individual server when idle, allowing for a more direct comparison with our proposal.

We consider the following parameters: Target CPU utilization for the low priority requests of 15% (C^* in the model), time window over which the CPU utilization is measured is 1 minute (t_w in the model), and sampling period is 5 seconds (t_s in the model). Thus, the proposed controller repeatedly calculates, based on feedback values received every 5 seconds, the maximum request rate that can be allowed so that the CPU utilization does not violate the target.

Two different scenarios are considered for the evaluation and comparison of the load balancing mechanisms: infinite load and limited load for low priority requests. In the infinite load scenario, the client makes low priority requests as fast as possible: a single message with a large number of requests is sent to the load balancer and, as soon as the response message is received, another message with many requests is sent. This scenario evaluates the controller response more directly, since there is always enough demand to achieve different values of CPU utilization. In the limited load scenario, the client periodically sends a message with a limited number of requests in a rate that is not enough to reach the target CPU utilization when the servers are idle.

4.1. Infinite load scenario

In this scenario we start at time $t = 0$ with the two servers idle (no high priority traffic), and thus receiving and processing low priority requests. At time $t = 4$ minutes, a high priority demand arrives to Server 0, generating a CPU utilization of 25%, which lasts until time $t = 9$ minutes. Server 1 never receives any high priority demands.

Figure 3 shows the behavior of the two servers under the modified Round Robin policy. Note that the computed maximum request rate keeps the CPU utilization of the servers on target when servers are idle (until $t = 4$ minutes), balancing the requests equally among the two servers (note that incoming traffic to each server is 25Kbps). However, there is no reaction to the high priority demand, at time $t = 4$ to Server 0, and the load balancer maintains the request rate to both servers. Thus, it continues to impose a CPU load on Server 0 with low priority requests while high priority demand is processing and generating CPU utilization above the target. When high priority load finishes at time $t = 7$ minutes, the servers return to the target CPU utilization.

Figure 4 shows the behavior of the same scenario with the proposed load balancing controller. Note that until time $t = 4$ minutes the behavior is very similar to round robin policy, with the controller meeting the target CPU utilization and equally dividing the requests among the two servers (note incoming traffic to each server). However, when high priority demands arrive to Server 0, the controller reacts by reducing the low priority traffic, allowing the CPU to essentially exclusively handle the high priority demand, which demands 25% CPU utilization. Note that low priority traffic to Server 0 reduced from 25Kbps to 5Kbps. When high demand is over, at around time $t = 8$ minutes, the controller ramps up the low priority request rate to meet the target CPU utilization, at 15%.

As expected, the controller does not increase the request rate of Server 1, in order to compensate the the low request rate sent to Server 0. Note that Server 1 is already at its target CPU utilization, and can thus not handle a higher request rate. As a consequence, the client must wait more, as requests previously handled by Server 0 are not being dispatched. Indeed, Figure 5 shows the network traffic leaving the client. At time $t = 4$ we

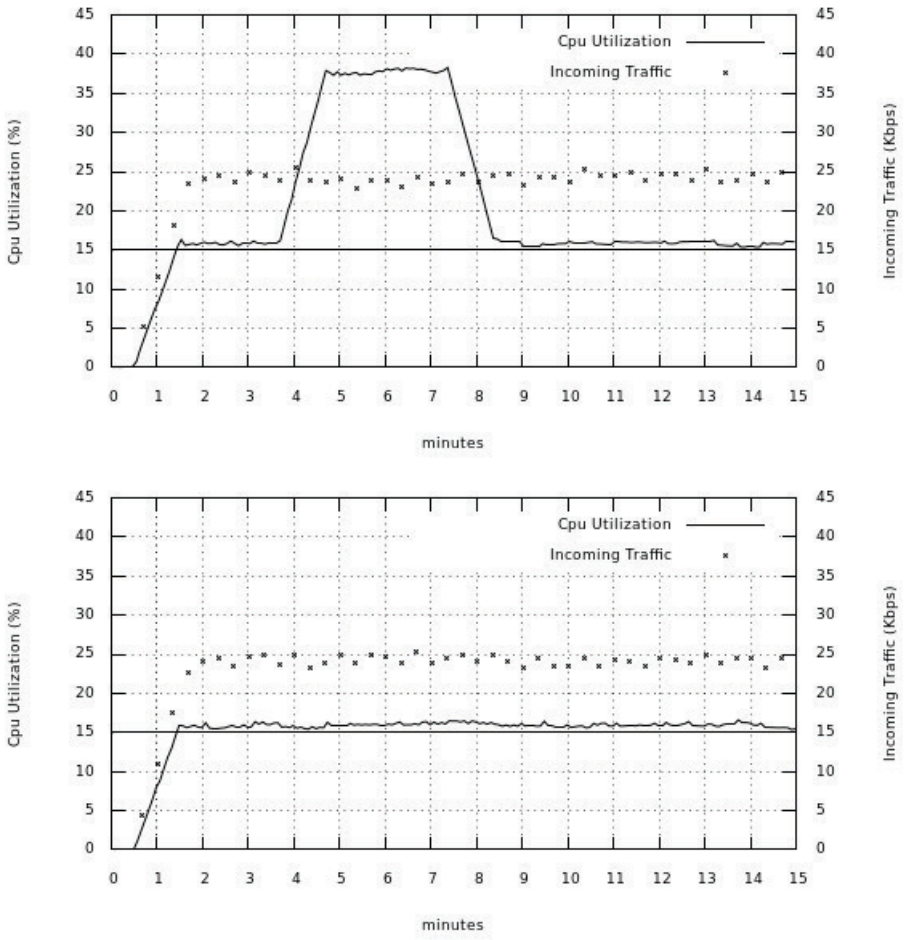

Figure 3. CPU utilization and network traffic on Server 0 (top) and Server (bottom) under modificed round robin policy for the infinite load scenario. At time $t = 4$ minutes high priority demand arrives in Server 0 that lasts until $t = 7$ minutes.

note that the delay between requests leaving the client increases (recall that client run an infinite request loop). At around time $t = 10$ the delay between requests return to original values. Thus, the proposed controller is effectively shielding the servers, pushing back to the client the dynamic adjustments in the request rates.

4.2. Limited load scenario

In this scenario the client application generates low priority requests periodically (at a fixed rate) such that the CPU load imposed on the servers is below the target of 15% (when servers are idle). However, this CPU load surpass the target when high priority demands arrives to the servers. Again, at around $t = 3$ minutes, high priority demand arrives to Server 0, generating a CPU load of about 25%, which finishes at around $t = 8$ minutes.

Figure 6 shows the behavior of the round robin policy, which as expected does not react to the sudden increase in load. The policy continues to dispatch requests to both servers equally, demanding CPU resources from Server 0, and thus reducing the CPU available to process the high priority demand.

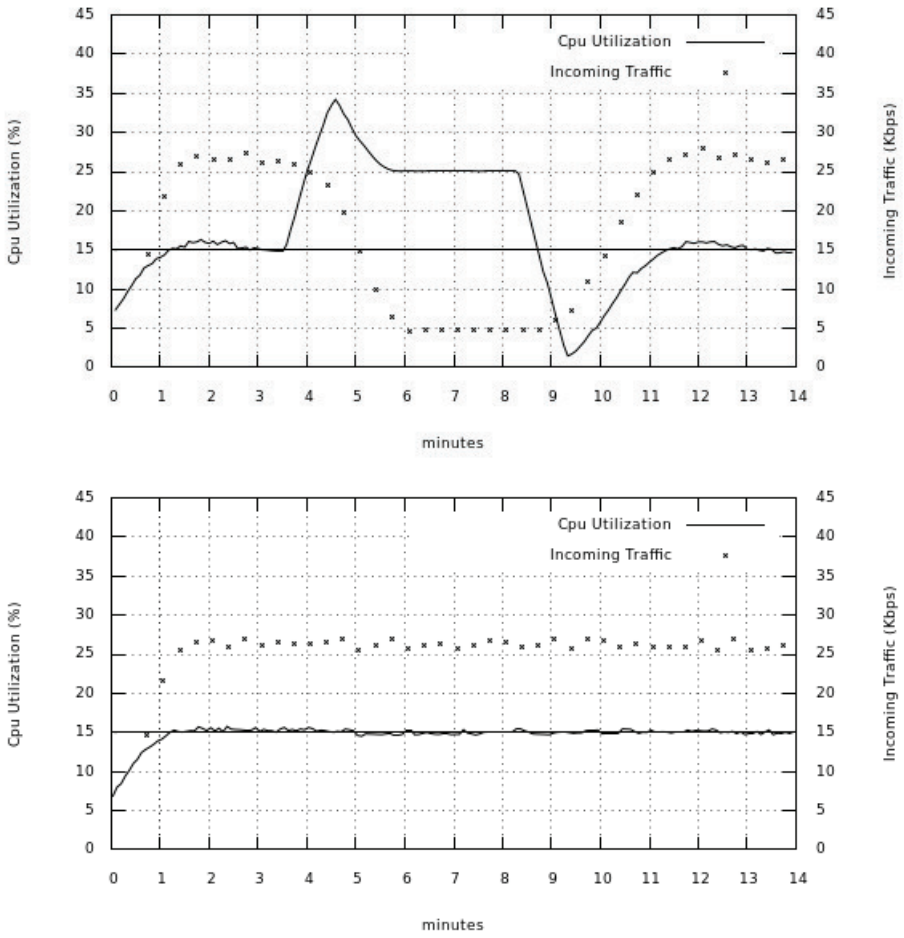

Figure 4. CPU utilization and network traffic on Server 0 (top) and Server (bottom) under proposed controller for the infinite load scenario. At time $t = 4$ minutes high priority demand arrives in Server 0 that lasts until $t = 7$ minutes.

As expected, the proposed mechanism behaves quite differently, as shown in Figure 6. In particular, the controller reacts to the increase in load in Server 0 by decreasing its low priority request rate, making available the needed CPU resources to process the high priority demand. Moreover, as the CPU utilization of Server 1 is still below the target, the load balancer increases the low priority request rate to that server. As a result, Server 1 absorbs the load that cannot be handled by Server 0, while meeting its target CPU utilization (of 15%). Note the increase in the network traffic to Server 1. At time $t = 9$ when high priority demand on Server 0 finishes, the controller again adapts and places part of the low priority request rate from Server 1 back to Server 0, equally distributing the load.

Interestingly, the reallocation of the client request rate from Server 0 to Server 1 allows the client to maintain its original request rate. Figure 8 shows that the client traffic remains unchanged while load is shiftted from Server 0 to Server 1 by the load balancer. The client is totally shielded from such dynamics on the servers, illustrating another benefit of the proposed controller.

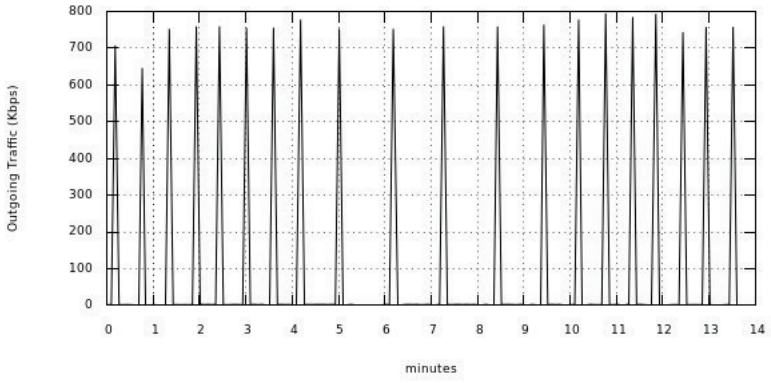

Figure 5. Outgoing network traffic in the client under the proposed controller for the infinite load scenario. Note that delay between requests increases at around $t = 4$ in response to high priority demands in Server 0.

5. Related Work

This paper considers the problem of distributing low priority requests across a set of servers while keeping CPU utilization imposed by such requests within a given target. This is necessary since servers must be ready to handle sudden high priority demands, not subject to the load balancer. Thus, our work is related to load balancing and to dynamic provisioning of resources, and in the following we comment on both research topics.

5.1. Load Balancing

Load balancing mechanisms are usually classified into static and dynamic. Static algorithms consider only nominal resources available on the servers, such as processing power and memory, but not any run-time characteristics, such as the actual load imposed on the server. The works of [Patel et al. 2013] and [Gandhi et al. 2015] present load balancing mechanisms to many different low-level contexts, such as NAT and Virtual IP Addresses. However, their proposal uses a simple static algorithm, known as *weighted random*.

The Weighted Least Connection (WLC) algorithm is a good example of dynamic load balancing since it sends arriving load to the server with smallest number of connections. However, this number may not accurately reflect the actual load on the server [Al Nuaimi et al. 2012]. Thus, [Ren et al. 2011] propose an improvement to the WLC algorithm, which considers a metric mixing usage of CPU, network, memory and disk. The algorithm then, having measured past values for such resources, predicts the next value for each server and sends the load to the one with the smallest value.

These algorithms are only marginally related to the our proposal, as they all consider a very different problem. In our scenario, we must decide not only which server is the best candidate for an arriving request, but also if any of the servers should receive the request at all! The servers may be too busy handling high priority demands and thus the load balancer may have to block the low priority requests. In other words, we perform not only load balancing, but also flow control. In this sense, our scenario and proposal also has similarities with dynamic resource provisioning.

[t]

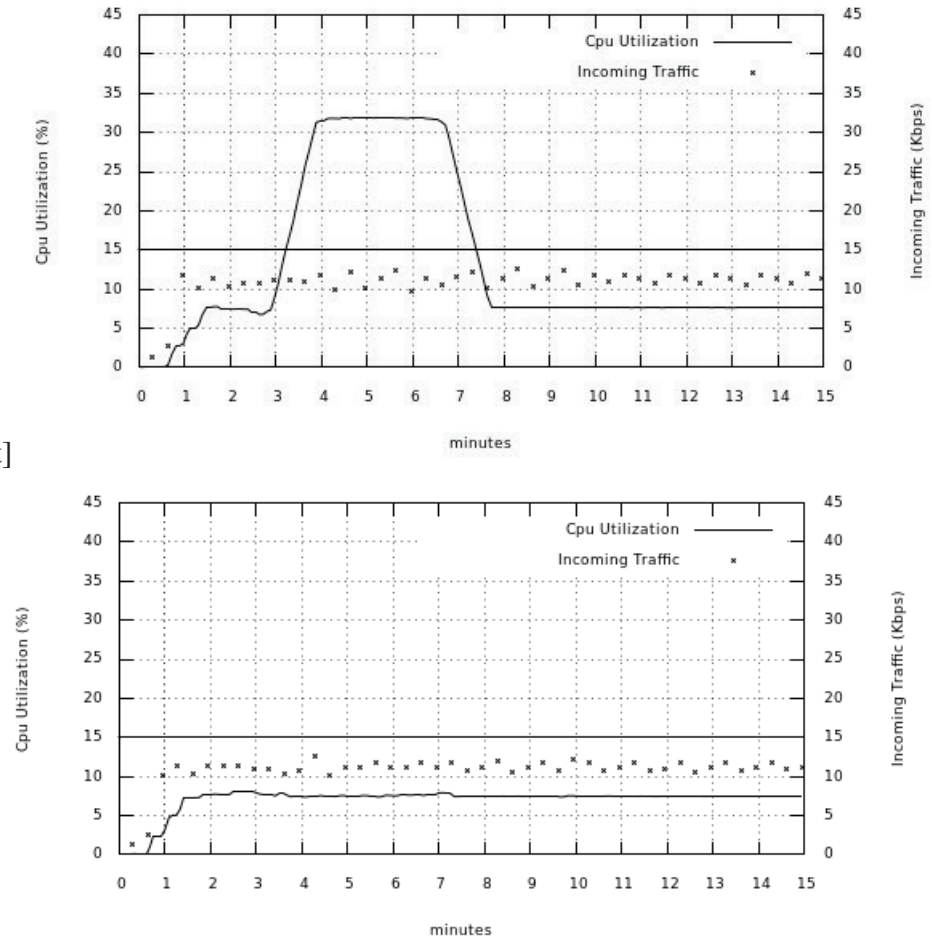

Figure 6. CPU utilization and network traffic on Server 0 (top) and Server (bottom) under modified round robin policy for the limited load scenario. At time $t = 4$ minutes high priority demand arrives in Server 0 that lasts until $t = 8$ minutes.

5.2. Dynamic Resource Provisioning

Dynamic resource provisioning is a highly researched subject nowadays, triggered mostly by large Cloud data centers and Big Data processing. In order for user applications to scale properly, more resources must be allocated to them when their load increases. However, the allocation should also be cost-effective and should not allocate more resources than needed. Thus, a lot of research has been done on automatic scaling techniques which can guarantee the SLA (Service Level Agreement) for the user applications cost-effectively for both the user and the Cloud provider. We comment on such mechanisms that leverage feedback/feedforward control methods, as our work also uses this methodology.

The work of [Dutreilh et al. 2010] mentions the difficulties of dynamic resource provisioning techniques based on thresholds or reinforcement learning when applied to real systems. They emphasize the fact that dynamic resource provisioning turn back to core concepts of automatic control - controllability, inertia, gain and stability. They summarize learned lessons mentioning good practices to be followed by automatic control algorithms - for example, that the time delay between two decisions must be long enough for the performance to stabilize to its new expected level, as we mentioned in Section 3.

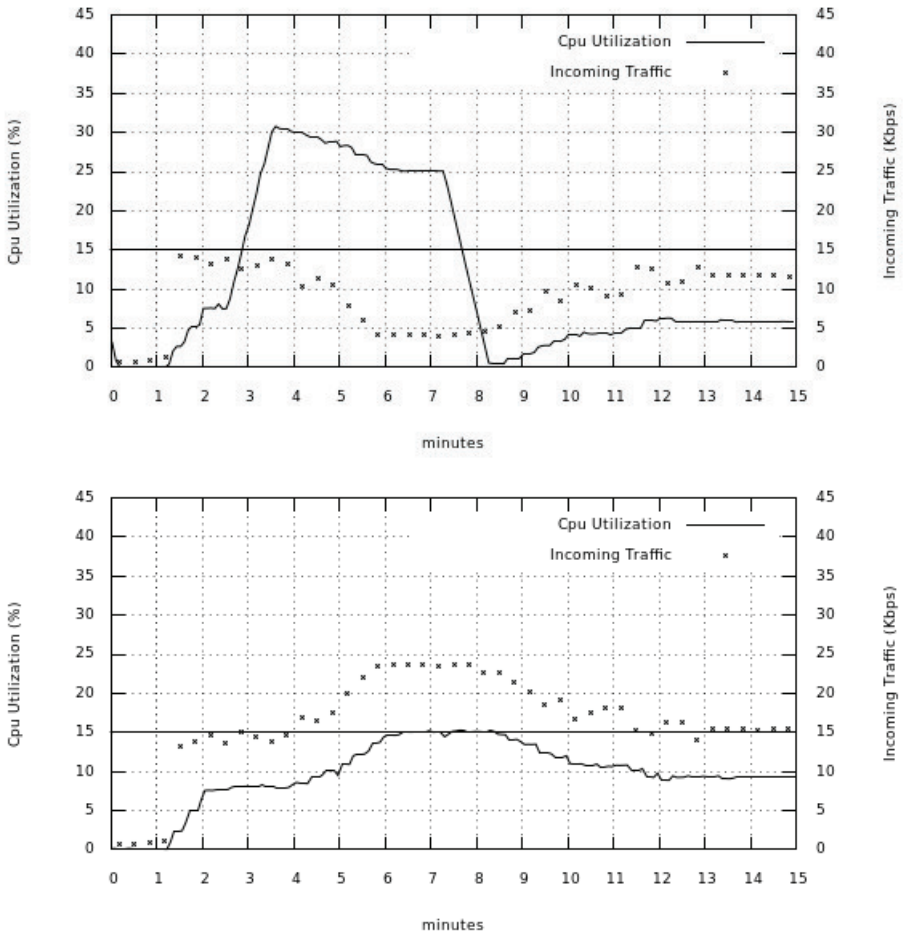

Figure 7. CPU utilization and network traffic on Server 0 (top) and Server (bottom) under the proposed controller for the limited load scenario. At time $t = 4$ minutes high priority demand arrives in Server 0 that lasts until $t = 8$ minutes.

Many papers have proposed dynamic provisioning algorithms for Cloud resources based on classical control theory. [Leontiou et al. 2010] propose a controller which uses a Proportional-Integral feedback and an adaptive feedforward methods to guarantee the SLA of general Cloud services. [Xiong et al. 2011] propose an adaptive PI controller to control the mean round trip time for N-tier web applications. [Ashraf et al. 2012] propose a proportional-derivative algorithm for scaling the application server tier and have deployed a prototype implementation in the Amazon Elastic Compute Cloud. The work of [Al-Shishtawy and Vlassov 2013] proposes a controller which mixes classical PI feedback with a feedforward controller to scale horizontally a cluster of key-value datastore and evaluate their algorithm on a Voldemort deployment.

Other researches, while not proposing classical control algorithms, have also based their proposal on feedback control theory. The work of [Zhu and Agrawal] proposes an algorithm based on an adaptive feedback controller for the allocation of cloud resources. The work of [Berekmeri et al. 2016] proposes a control algorithm based on classic feedback and feedforward methods to scale automatically the number of nodes in a MapReduce cluster in order to guarantee a certain level of performance.

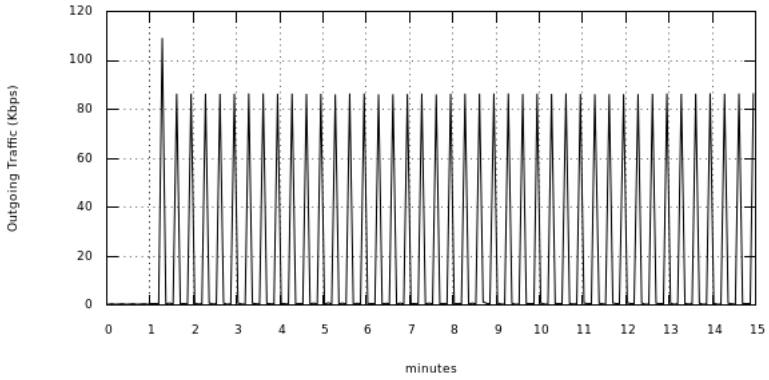

Figure 8. Outgoing network traffic in the client under the proposed controller for the limited load scenario. Note that client request rate remains unchanged despite the significant decrease in the request rate to Server 0.

The papers above assume there is always more available resources that can be allocated to an application in order to maintain its SLA. This is a reasonable assumption in many cases. However, our problem considers a small pool of resources which must be shared between applications with different priorities, and when a resource is allocated to handle a request, it immediatly interferes with the other requests. It is also worth noting in our scenario only low priority demands are subject to control, with high priority demands arising arbitrarily to servers.

6. Conclusions

This paper presented a load balancing mechanism based on closed-loop controller for a scenario where low priority requests traverse the load balancer while high priority demands can suddenly arrive directly at the servers. In particular, beyond distributing the load among the servers, the load balancer keeps CPU utilization arising from low priority requests within a specified target.

A real implementation of the controller has been developed and used to evaluate two different scenarios that illustrate that typical load balancing policies, such as round robin, are not appropriate in this context. In contrast, the proposed mechanism was shown to successfully adapt to dynamic changes to high priority demands, reducing or increasing the load imposed by low priority requests, meeting the target CPU utilization.

In the specific scenario considered, CPU utilization was measured by a one-minute moving average and thus the controller response had timing parameters of the same order. In some situations, a one-minute time constant may not be sufficient, but that is not a limitation of the proposed load balancing mechanism. If CPU utilization is measured using a smaller time-window, then it is possible to obtain a controller with faster response using the same design we proposed, by simply properly adjusting the time constants.

Finally, as the design of the controller used by the balancer needs tuning the value of some constants, a future work is to automatically determine these constants using an online regression with recursive least squares for example, establishing a relationship between the variation of measured feedback values to the variation of the control signal.

References

Al Nuaimi, K., Mohamed, N., Al Nuaimi, M., and Al-Jaroodi, J. (2012). A survey of load balancing in cloud computing: Challenges and algorithms. In *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*, pages 137–142. IEEE.

Al-Shishtawy, A. and Vlassov, V. (2013). Elastman: autonomic elasticity manager for cloud-based key-value stores. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 115–116. ACM.

Ashraf, A., Byholm, B., Lehtinen, J., and Porres, I. (2012). Feedback control algorithms to deploy and scale multiple web applications per virtual machine. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 431–438. IEEE.

Berekmeri, M., Serrano, D., Bouchenak, S., Marchand, N., and Robu, B. (2016). Feedback autonomic provisioning for guaranteeing performance in mapreduce systems. *IEEE Transactions on Cloud Computing*.

Dutreilh, X., Moreau, A., Malenfant, J., Rivierre, N., and Truck, I. (2010). From data center resource allocation to control theory and back. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 410–417. IEEE.

Gandhi, R., Liu, H. H., Hu, Y. C., Lu, G., Padhye, J., Yuan, L., and Zhang, M. (2015). Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38.

Leontiou, N., Dechouniotis, D., and Denazis, S. (2010). Adaptive admission control of distributed cloud services. In *2010 International Conference on Network and Service Management*, pages 318–321. IEEE.

Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D. A., Kern, R., Kumar, H., Zikos, M., Wu, H., et al. (2013). Ananta: cloud scale load balancing. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 207–218. ACM.

Ren, X., Lin, R., and Zou, H. (2011). A dynamic load balancing strategy for cloud computing platform based on exponential smoothing forecast. In *2011 IEEE International Conference on Cloud Computing and Intelligence Systems*, pages 220–224. IEEE.

Xiong, P., Wang, Z., Malkowski, S., Wang, Q., Jayasinghe, D., and Pu, C. (2011). Economical and robust provisioning of n-tier cloud workloads: A multi-level control approach. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 571–580. IEEE.

Zhu, Q. and Agrawal, G. Resource provisioning with budget constraints for adaptive applications in cloud environments. *IEEE Transactions on Services Computing*, 5(4):497–511.