Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia

COPPE UFRJ

A CONTROL-BASED LOAD BALANCING ALGORITHM WITH FLOW CONTROL FOR DYNAMIC AND HETEROGENEOUS SERVERS

Rodolpho Guedino de Siqueira

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Daniel Ratton Figueiredo

Rio de Janeiro
Julho de 2017

A CONTROL-BASED LOAD BALANCING ALGORITHM WITH FLOW CONTROL FOR DYNAMIC AND HETEROGENEOUS SERVERS

Rodolpho Guedino de Siqueira

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

RIO DE JANEIRO, RJ – BRASIL
JULHO DE 2017

Siqueira, Rodolpho Guedino de

A Control-based Load Balancing Algorithm with Flow Control for Dynamic and Heterogeneous Servers/Rodolpho Guedino de Siqueira. – Rio de Janeiro: UFRJ/COPPE, 2017.

XII, 39 p.: il.; 29, 7cm.

Orientador: Daniel Ratton Figueiredo

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2017.

Bibliography: p. 38 – 39.

1. Load-balancing. 2. Feedback Control. 3. Distributed Systems. 4. Computer Networks. I. Figueiredo, Daniel Ratton. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

I dedicate this work to those who
have dedicated their life to me -
my parents: Marco and Tânia;
Antônio and Elydea.

Acknowledgements

I would like to thank Prof. Daniel Figueiredo for all the valuable help he provided me. He has taught me so much and has also been very comprehensible during our work together. I would also like prof. Daniel and prof. Valmir Barbosa for giving me opportunity to attend their classes, which were sincerely some of the best I have had in my life. Finally, I would like to thank prof. Lucia Drummond for the help and availability to review this work. Thank you all very much!

Then I will mostly repeat the people I have already thanked in other occasions and who I will probably always thank in my life:

I want to thank the friends that I consider to be my second family: Felipe Alecrim, Gilson Falcão, Hugo Lins and Thiago Girard. There is no way I can properly thank all that I have gone through with you guys. You made my life in Recife so much better and easier, and it was with you that I spent most of the best moments of my life! I am sure there is more to come, and I will be there for you wherever I am.

I also thank my great friends Fernando Rodrigues and Renan Ferraz for the amazing friendship and all awesome conversations we have had. I really like you guys and I look forward to having you as my best men very soon at my wedding!

I thank all my cousins, for they have been my friends all my life, and especially the older ones for being such role-models. I thank my cousin José Jr. for his close friendship during all our lives, as we have grown up together and have always supported each other.

I greatly thank my mom for her lighthearted spirit, which really inspires me to less serious, for the strength she has shown to me throughout life and for all the incredible food! She is one of the most talented persons I have known. I also greatly thank my dad for all the lessons, for the impressive clearness of thought and all the braveness shown during some more difficult time. Everything I am is because of you two, and I am so proud of it!

I thank my stepmom, Ana, for being so good to me all this time and for giving me, along with my dad, one of the true joys of my life: my sister Lívia. I also thank another true joy of my life: my little brother, João. I am so proud of you, my brother. You have a brilliant future ahead of you!

I thank my grandparents, Elydea and Antônio, for all the effort and all the help. I also thank them for the good lessons on life and marriage and for providing me such good role-models on how to be a man: my dad, my uncle and grandpa himself. I miss you so much, grandpa! So much!

I would like to thank my fiancée Mayara Llaguno for the comprehension during stressful times, for all the love, laughter, good food, interesting conversations and awesome dancing! I also thank you for helping me achieve everything we now have together. I love you so, so much! You're my princess! I would also like thank my in-laws, Silvio, Elaine and Caique Llaguno, as they have become a real family for me.

At last, for this and any work of science I will always especially thank my dad, for all the answers he gave me, and my little brother, for all the questions he made me. They are indeed the best science teachers I have had. They are the ones who have never let the discovery spirit vanish in me.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UM ALGORITMO DE BALANCEAMENTO DE CARGA BASEADO EM CONTROLE DE FLUXO PARA SERVIDORES DINÂMICOS E HETEROGÊNEOS

Rodolpho Guedino de Siqueira

Julho/2017

Orientador: Daniel Ratton Figueiredo

Programa: Engenharia de Sistemas e Computação

Apesar de o problema de balanceamento de carga ser fundamental e ter sido bem estudado como problema de alocação de recursos, os cenários e tecnologias diferentes que sempre surgem em sistemas distribuídos demandam novas abordagens e algoritmos. Nesse contexto, nós consideramos um problema prático de um cenário real no qual os servidores são heterogêneos e sujeitos a uma carga de fundo não controlada pelo balanceador de carga. Nesses casos, políticas clássicas tais como Round Robin ou mais novas como Join the Shortest Queue não são efetivas. Nós propomos um algoritmo de balanceamento de carga que despacha requisições a um conjunto de servidores heterogêneos de acordo com a sua disponibilidade de recursos de CPU, utilizando controle com retroalimentação para prevenir sobrecarga. Nós implementamos essa política e avaliamos sua performance em um cenário real controlado. Nossa avaliação indica que o algoritmo proposto é mais eficaz em distribuir a carga do que outras políticas clássicas nesse cenário, em particular quando a carga de fundo é dinâmica.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A CONTROL-BASED LOAD BALANCING ALGORITHM WITH FLOW CONTROL FOR DYNAMIC AND HETEROGENEOUS SERVERS

Rodolpho Guedino de Siqueira

July/2017

Advisor: Daniel Ratton Figueiredo

Department: Systems Engineering and Computer Science

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 the novel join-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 this scenario, in particular when background load is dynamic.

Contents

List of Figures

List of Tables

Chapter 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 [1].

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 [2], as different scenarios and assumptions emerge from technological advancements and new applications.

This work considers a real world scenario where common assumptions do not hold, prompting the design of a new load balancing mechanism.

A 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 our case, the balancer and the clients of the application handled by the balancer are far away from the servers.

Another common assumption in the design of load balancing mechanisms is that servers receive requests (i.e., load) only from single load balancer. In this case, the load imposed on the servers comes only from one source, and hence no feedback mechanism is needed in order to evaluate the availability of each server. However, there are practical scenarios in which the servers are subject to load from different applications and the total load is not handled by a single load balancer. In these cases, even if there is a load balancer for each application, they do not know the load caused by each other, and hence cannot make decisions regarding the availability of resources on the servers without a feedback mechanism or some kind of communication among them.

One scenario in which it would be very difficult to create a single load balancer for all load is one in which there is at least one application that uses a proprietary protocol over TCP. In this case, there is not a simple way to interpret the application's TCP stream and split it into well defined requests that can be dispatched between servers.

Another hard case for load-balancing is one in which the servers are utilized for a stateful application. In this case, when a client connects to a server, no balancing can be done with his following requests, because servers do not have their full state synchronized. Thus, a client connects to a server and sticks to it for a long period, and its requests are handled as if there was no load balancer.

Our case is a mixture of the two above scenarios, with servers handling load from two different applications, with the high priority application being stateful and using proprietary protocol. Thus, the high priority requests are not handled by our load-balancer. On the other hand, there is a low priority stateless application which uses a known protocol, and thus we create a load balancer to dispatch those requests between the servers in a controlled way, in order to not overload them, leaving enough resources available for the high priority application.

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 also run a mission critical application that has high priority and unpredictable behavior - the application remains idle for long periods and suddenly needs to be served with no delay. 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 1.1 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.

For the reasons mentioned above, the high priority requests cannot be handled by the load balancer used for the low priority ones. Thus, as servers are subject to load not controlled by our load balancer, the decision of which server will receive the requests should consider real-time information regarding the availability of resources for each server.

As the servers in our scenario run legacy operating systems which do not provide

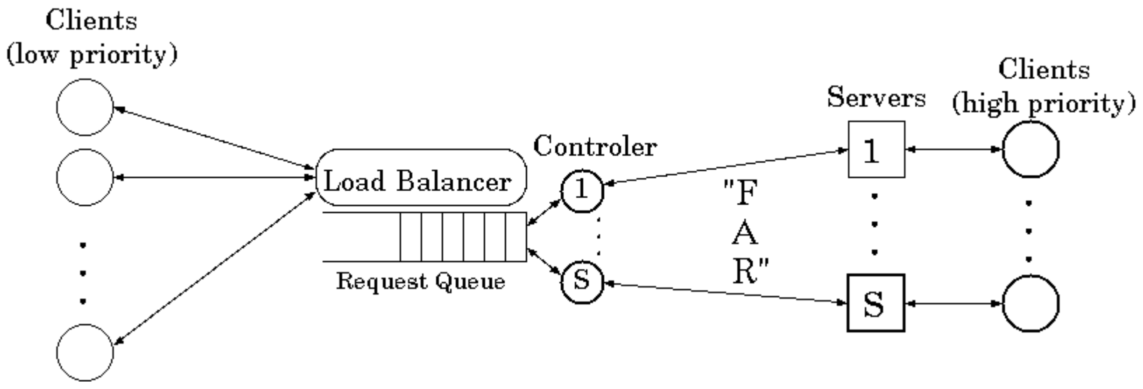


Figure 1.1: Illustration of the considered scenario

many tools for programming complex software, we choose make the server software as simple as possible and decide to leave to a centralized load balancer all the complexity of the balancing decisions. Hence, given that premise, strategies such as Join-Idle-Queue, which require the servers to actively inform the balancer when they are idle [3], are not a good choice.

Considering that the servers do not perform any active role in the balancing decision and that all decisions are centralized in the load balancer but not all load passes through it, we need some kind of polling measurement which can then be used in a feedback control mechanism.

For this scenario, we propose a load balancing mechanism based on a Integral Controller, which is a variation of a very common algorithm in Control Theory, vastly used in the Industrial Automation field, the PID. Both our algorithm and the PID calculate their control decision as a function of the error, defined as the difference between the current value of the variable being controlled - in our case, the CPU utilization - and its desired value. The full PID variation calculates its output as the sum of three functions of the error: one is proportional to the current error, the other is proportional to the sum of the past measured errors and the last is proportional to the current derivative of the error - hence the name PID: Proportional, Integral, Derivative controller. Our controller uses only the integral part.

Although very common in Industrial Automation for controlling Process variables such as temperature and pressure, feedback control algorithms have not been so vastly used in Computer Science up to the last years. However, in order to reduce costs, delays and increase efficiency, Cloud providers nowadays have the need to fully automate the tuning and operation of their platforms [4], and thus the interest in automated feedback control systems has increased in this industry [5, 6]. Some applications of a feedback control system in Computer Science nowadays include automatically adjusting the following: cache size by measuring the hit rate;

number of deployed servers by measuring request latency; number of worker threads by measuring wait time.

This work proposes an effective mechanism for the above scenario in which the load balancer dispatches aggregated requests to the servers according to a rate determined by a corresponding feedback control loop 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 dissertation is organized as follows. Chapter 2 poses the problem and the challenges in devising a load balancing mechanism in this context. Chapter 3 reviews the relevant work related to load balancing and control algorithms within the context of Dynamic Resource Provisioning. Chapter 4 describes the feedback controller for the proposed load balancing mechanism, while chapter 5 presents the details of the real-world deployment of the proposal. Chapter 6 discusses the results of the test implementation when running in different scenarios. Finally, chapter 7 presents our conclusions and possible future work.

Chapter 2

Problem Formulation and Objectives

As discussed in chapter 1, we consider a real world scenario in which a small set of servers must respond with high priority to a mission critical application whose behavior is not easy to predict, but that remains 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, with the condition that this monitoring has low priority with respect to other applications running on them. 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, as variables are consistently 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, this kind mechanism does not necessarily distribute the load proportionally to the available resources (i.e., CPU) - even overloaded servers would keep receiving the same share of total load.

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. In the case when even the most available server should not receive further requests because it is operating with a CPU utilization above the pre-specified target, the balancer could totally refrain from sending requests. However, this kind of policy may lead to oscillation between two extremes, full-rate or no flow. In this case, when a server becomes available it can receive a great amount of flow and suddenly become overloaded. Instead, we choose to control the load in a more subtle way, by controlling the flow to a steady target.

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 these 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.

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.

Chapter 3

Related Work

This dissertation 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.

3.1 Load Balancing

The work of [7] provides a good survey of the available load balancing algorithms. We use that survey as a reference and comment on the most relevant algorithms below.

Load balancing algorithms can be basically classified into two classes, static and dynamic. Static algorithms consider only nominal resources available on the servers, such as processing power and memory, while dynamic algorithms consider run-time characteristics, such as the actual load imposed on the server - as is considered by our proposal.

The main advantage of static algorithms is their simplicity and fast execution at run time, because all data considered is available beforehand and no complex operations are needed when the system is running. However, they do not adapt to the system dynamics.

Dynamic load-balancing algorithms have the advantage that they can adapt to changes in load or to changes in the availability of resources of each server at run time, but this comes at the expense of extra communication and/or computation overhead.

The dynamic load balancing algorithms can be centralized, decided by a single balancer, or distributed, involving the participation of many different processors. Centralized approaches are very simple, but suffer from bottleneck and have a single point failure. Distributed load balancing algorithms are free from these problems, but are more complex and sometimes require non-trivial communication among participants.

Distributed dynamic load-balancing can be cooperative or non-cooperative. In the latter type, a balancer makes decisions autonomously, without taking into consideration the rest of the balancers decisions. In cooperative systems, the balancers act as a team and each one has the responsibility to execute its own part in order for the whole system to achieve its global goal.

We comment on some of those algorithms below, focusing on centralized load-balancing systems, because these are more related to our scenario.

3.1.1 Static Algorithms

A Randomized algorithm randomly selects a server to process a request. It is a very simple algorithm, but it does not consider the availability of resources of the server being selected. This may result in the overloading of servers and unnecessarily long response times. The works of [1] and [2] present load balancing mechanisms to many different low-level contexts, such as NAT and Virtual IP Addresses using a *weighted random* approach.

The Round Robin algorithm assigns requests to servers in a rotating manner. The load balancer starts with any random server and the subsequent requests are then served in a circular order. If the requests are not homogeneous, some servers may become much more loaded than the rest.

Weighted Round Robin is a modified version of Round Robin which considers the differences in nominal characteristics of the servers. If a server has twice the processing power of the others, the load balancer will assign two requests to this server for each request assigned to the weaker ones. It suffers from the same problem as the original Round Robin - if requests are heterogeneous, a server may become much more loaded than the others.

3.1.2 Dynamic Algorithms

Active Monitoring Load Balancing (AMLB) Algorithm maintains information about each server and the number of requests currently allocated to them. When a request arrives, the balancer chooses the least loaded server. As the choice of server only considers its current load and not its processing power, the processing time of some requests may suffer and violate QoS requirements.

Weighted Active Monitoring Load Balancing Algorithm is a combination of Weighted Round Robin and Active Monitoring Load Balancing Algorithm. In this algorithm different weights are assigned to the servers, considering their processing power capabilities. The requests are assigned to the least loaded servers according to their capabilities. It thus tries to solve the disadvantage of the Active Monitoring Load Balancing Algorithm by considering also the processing power of available servers.

Join-Idle-Queue is an algorithm proposed for dynamically scalable web services. This algorithm provides large scale load balancing with distributed dispatchers by removing the load balancing work from the critical path of request processing. It decouples the discovery of lightly loaded servers from job assignment, so that a balancer, when receiving a request, already knows the available servers beforehand. This is done as the idle servers inform the balancers when they become idle. This removes the load balancing work from the critical path of request processing [3]. This algorithm is proposed to solve problems in distributed load balancing scenarios with many balancers, which makes it more complex, because the servers need to decide which balancer(s) to inform. However, as our scenario deals with a centralized balancer, we will not comment on those details.

The Weighted Least Connection (WLC) algorithm sends load to the server with the smallest number of connections. However, this number may not accurately reflect the actual load on the server and the algorithm does not take into consideration the capabilities of the server, such as storage capacity, processing power and bandwidth [8].

Exponential Smoothing Forecast based on Weighted-Least Connection (ESB-WLC) is proposed by [9], an improvement to the WLC algorithm which considers a metric mixing usage of CPU, network, memory and disk. The algorithm, 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. ESBWLC takes advantage of all historical data and, based on exponential smoothing, distinguishes them through the smoothing factor to let recent data make a greater impact on the predictive value than long-term data.

Join-Shortest-Queue algorithm sends an arriving request to the server with the smallest queue length. The work [5] proposes a load balancer which modifies JSQ and which would work for brownout applications, where servers have the ability to provide as response only part of the content being requested, in order to adapt and to guarantee response times when the load is high. They propose a modification by adding an offset to the queue length of the servers based on their current length. They actually define which server should have the virtual shortest queue length, resulting from the sum of the actual queue length with the offset, which will then be chosen by the standard JSQ algorithm. For the offset calculation, they propose a PI controller, which reads the current length and uses it as the feedback measurement. Thus, they use the offset as the control action and the queue length as the controlled variable. Their solution is general enough so that non-brownout applications are seen as a particular case and can also be handled by the load balancer. The only difference made for the brownout case is that the controller takes into consideration not only the queue length but other measurements related to the brownout algorithm being utilized. In the non-brownout case, their PI controller considers only the queue length and in fact becomes an integral-only controller, like the one we propose, but they only provide empirical values for their tuning.

These algorithms are related to the our proposal, but they all consider a 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, which provides or takes resources from an application depending on the state of the other. In this sense, our scenario and proposal also has similarities with dynamic resource provisioning.

3.2 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. Still, many current commercial services only provide elasticity mechanism with thresholds and parameters left to the users, who most probably cannot make those choices optimally and are usually more focused on the application details.

The work of [10] 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 mention in chapter 4.

Automated control for elastic storage is the subject of the research in [11]. They specifically deal with common problems in control applications, such as actuator delays and interference in measured signals. A classic integral controller is chosen as their control policy - very similar to our choice-, with the addition of a dead band around the setpoint to deal with the oscillation which may be caused by the discrete actuator.

Many papers have proposed dynamic provisioning algorithms for Cloud resources based on classical control theory. The work of [12] proposes a controller which uses a Proportional-Integral feedback and an adaptive feedforward methods to guarantee the SLA of general Cloud services, while [13] propose an adaptive PI controller to control the mean round trip time for N-tier web applications. A proportional-derivative algorithm is proposed by [14] for scaling the application server tier and have deployed a prototype implementation in the Amazon Elastic Compute Cloud. The work of [15] proposes a controller which mixes classical PI feedback with a feed-forward 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 [16] proposes an algorithm based on an adaptive feedback controller for the allocation of cloud resources. The work of [17] 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. They propose a dynamic model that predicts MapReduce cluster performance based on the number of nodes and the number of clients and, based on this model, they propose a controller based on classic feedback and feedforward methods.

The papers above, however, 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, but 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 immediately interferes with the other requests. This is the main difference between this work and the current literature. It is also worth noting that in our scenario only low priority demands are subject to control, with high priority demands arising arbitrarily to servers. Last, we should note that the majority of the control-based proposals we mentioned use empirical tuning values with no insight as to how they can be obtained, while we try to show how the tuning must be changed if the certain conditions of the problem vary.

Chapter 4

Proposed Load-balancer

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.

4.1 Balancing mechanism

The balancer has a single request queue, from which requests are taken, bundled and sent to the servers (see Figure 1.1). The queue is accessed in parallel by different threads, one for each server.

The requests are bundled so that the network resources are more efficiently used and so that the long $RTT \approx 20$ ms does not limit the rate of request service.

The balancer sends bundles one at a time. When the balancer completes sending a bundle, it waits for the server to respond to all requests in that bundle before sending another one.

Since the flow of requests to a server at time instant t depends on both the number of requests N_t in a bundle and the interval between bundles, we could control the rate by varying either of those values.

We could try to control the flow of requests by choosing a fixed time period of t_r and by sending to a server bundles with a varying number of requests, N_t, every t_r seconds. This scheme would make the balancer start counting the elapsed time right after sending a bundle of requests so that, when receiving the responses, it could know how much to wait until t_r elapsed, allowing it to send another bundle. Considering that a request takes t_p seconds to be processed, this would work well if the time necessary to process all N_t requests in a bundle, $N_t t_p$, was smaller than t_r - the balancer would only need to wait a period equal to $t_r - N_t t_p$. However, as N_t varies, we could face problems when N_t increases and the time necessary to process all the requests becomes larger than t_r. If that happened, we would not be sending

N_t requests every t_r time units anymore. Thus, we could not control the flow of requests in this manner.

In order to have a known flow of requests to a server, we choose to vary the number of requests in a bundle, N_t, and we make the balancer wait for a fixed time period t_d after the arrival of the responses for the last bundle of requests: it waits for more t_d seconds after receiving the responses, and only then sends another bundle.

Thus, we control the flow of requests by varying N_t while using a fixed delay parameter t_d.

The number of requests per bundle that can be sent to each server, N_t, is defined by its own controller. The control is not performed jointly, but occurs independently for each server, following the same control algorithm.

4.2 Controller

In order to decide how N_t should vary for properly controlling the CPU utilization of a server, we propose a Integral Controller, a variation of the vastly used PID, Proportional Integral Derivative, using only the integral action.

Usually, control problems are basically split into two categories, disturbance rejection and command tracking. The goal of our balancer is the latter: to increase slowly the CPU utilization of the servers due to low priority requests until a budget is reached.

Once the flow to a server is at the target, we do not intend to actually keep the total CPU utilization regulated at that value at all times, because that would mean rejecting the "disturbances" caused by high priority load. If that was the goal, other kinds of controllers would be more appropriate, such as PI controller for example. However, in our case we could never do that kind of regulation, because high priority requests shall not be controlled at all and we have no power over them. So, our integral controller will actually react to "disturbances", but will not react to them as if it intended to quickly regulate the CPU utilization at all times.

An integral-only controller is sufficient for our goal and, due to its simplicity, this controller is easier to understand, model and operate, as it needs fewer parameters than the full PID variation or other more complex control algorithms. Moreover, it is also a goal of our work to show with simple models and intuitive reasoning how to find the parameters of the controller.

After designed, the load-balancer is going the need operational adjustments to tune the controllers when the characteristics of the servers change. Thus, simplicity and ease of understanding greatly facilitates its operation.

We thus use a discrete-time integral controller for each server, which varies the flow of requests served by updating its control decision variable N_t, the number of requests per bundle. The controller updates its decision every t_s seconds, after having a new feedback measurement of CPU utilization, C_t, trying to control its value at C^* :

$$N_t = N_{t-1} + g(C^* - C_{t-1}) \tag{4.1}$$

Then the only tuning we need to do is to choose properly the single parameter g, commonly called controller gain, which tells the controller how much to change its decision based on the difference between the target C^* and the current value of CPU utilization.

4.2.1 Modeling and Tuning

We propose a controller tuning 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. And every reference we make from now on to variables related to time shall be considered in the controller frame of reference.

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. Moreover, the controller also updates its control decision once every t_s seconds. The measurements are taken using the complement of a classic Linux administration metric, "CPU idle", which is calculated as a moving average over the last minute. Also, it was found experimentally that this metric does not change significantly in a period smaller than 5 seconds, so we chose $t_s = 5$ - which is much greater than $RTT \approx 20$ms. This measurements normally do not impose any significant load on the servers, and their impact is even smaller with this sampling period of many seconds.

If the controller decision-making period is smaller than t_s, then it will see the same previously measured value and may prematurely change its decision. On the other hand, if takes longer, it may unnecessarily delay its reaction to changes in CPU resources availability.

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 requests received with each bundle. 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 bundles.

Let C_t denote CPU utilization at time t, N_t the number of requests in a bundle

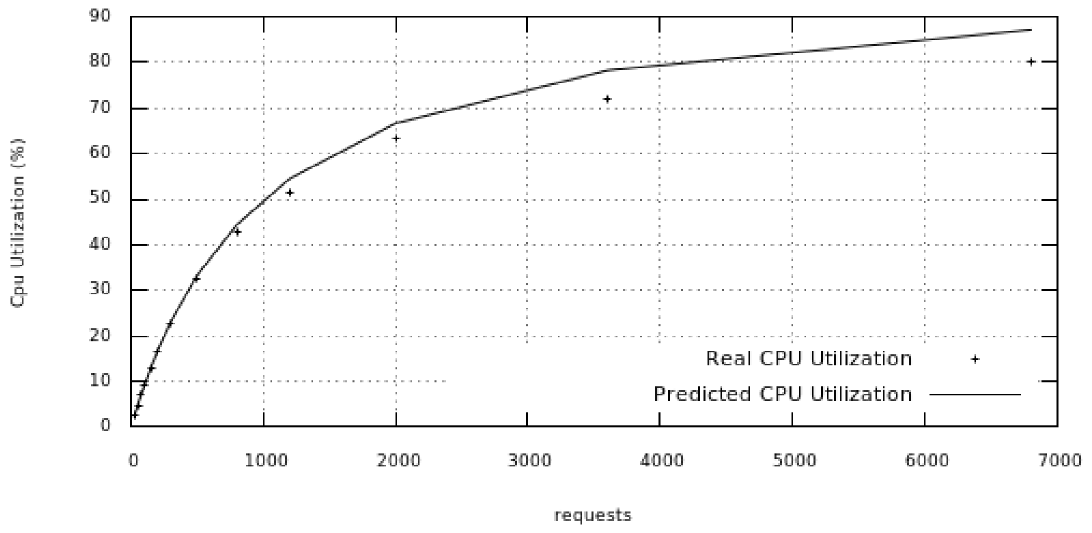


Figure 4.1: Comparison between model prediction and real system behavior (CPU utilization as a function of number of requests sent periodically).

at time t, t_p the time required to process a single request, and t_d the delay period the balancer waits after receiving the responses of the last bundle. With those definitions we have the following ratio:

$$C_t = \frac{N_t t_p}{t_d + N_t t_p} \tag{4.2}$$

where we have assumed that the network delay is negligible when compared to t_d and to the time required to process the whole 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 relating a change in the number of requests per bundle to a change in the measured CPU utilization, which we need to consider when tuning 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 4.1 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:

$$\begin{aligned} C_t - C_{t-1} &= (N_t - N_{t-1})\frac{dC_{t-1}}{dN_{t-1}} \\ &= (N_t - N_{t-1})\frac{d}{dN_{t-1}}\left(\frac{t_p N_{t-1}}{t_d + t_p N_{t-1}}\right) \\ &= (N_t - N_{t-1})\frac{t_p t_d}{(t_d + t_p N_{t-1})^2} \end{aligned} \tag{4.3}$$

Now, considering that C is measured by a moving average, we cannot expect to control C by making it vary the desired amount faster than the window over which the average is calculated, t_w, and this is considered by the tuning.

Our controller thus tries 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. In other words, the controller should take longer to achieve the reference if t_w is long, and make smaller variations to N_t.

Note also that, since the controller only updates its control decision when it receives a feedback sample, once every t_s seconds, the variation its control action 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:

$$\begin{aligned} C_t - C_{t-1} = N_t &= N_{t-1} + g(C^* - C_{t-1}) \qquad (4.4) \\ &= (N_t - N_{t-1})\frac{t_p t_d}{(t_d + t_p N_{t-1})^2} \end{aligned}$$

This yields a integral controller which, at each instant t taken discretely once every t_s seconds, increments the allowed number of requests in each bundle to a server based on a fraction of the difference between the reference CPU utilization, C^*, and its current measured value:

$$\begin{aligned} N_t &= N_{t-1} + k\frac{t_s}{t_w}\frac{(t_d + t_p N_{t-1})^2}{t_p t_d}(C^* - C_{t-1}) \qquad (4.5) \\ &= N_{t-1} + g_t(C^* - C_{t-1}) \end{aligned}$$

This controller is adaptive and non-linear, as it adjusts the fraction of the measured difference used for the control action increments depending on the current control action value. So, g in 4.1 is fact varying in time and becomes g_t - this adaptation is sometimes called gain-scheduling.

The parameter $k = 0.6$ was chosen experimentally in order to main the controller response stable. The experiments have shown that if $k > 0.6$, there is an overshoot of the target CPU utilization, which is not desired according to our goals.

Note that, although there is an experimental parameter in the proposed tuning, the modeling we have shown provides an intuitive way to evaluate how the tuning should vary if the characteristics of the servers change. In fact, chapter 6 shows that the controller dynamics work equally well in different scenarios if k is maintained constant and t_p varies to reflect the server processing power.

Finally, note also that this controller can be extended to handle different kinds of requests, each one with a different processing time, as long as these times are are known in advance by the balancer so that the t_p can be adjusted in real-time as requests are received. Moreover, the same mechanism can adjusted for controlling some metric related to IO resources instead of CPU utilization.

Chapter 5

Deployment in Real Scenario

The proposed load-balancer algorithm was implemented in a proxy actually deployed in a real scenario. This proxy is being used in the biggest Oil and Gas company in Brazil to provide access to a remotely distributed and replicated database for multiple real-time monitoring applications.

The load-balancing mechanism was implemented as part of a proxy which receives requests from the client applications and dispatches them to the available servers.

In summary, these are the services which are provided by the implemented proxy:

1. Caching - if two clients make the same request in a small period of time, the last client receives a copy of the response received by the the first client; in the same manner, if the same client repeatedly makes the same request during the cache expiration interval, it will receive a copy of the response obtained.

2. Request Grouping - as the clients are independent, they make their requests in an asynchronous way, and that could lead to many small packets being sent to the remote servers. Thus, the proxy waits for the the arrival of requests until a timer expires, groups the received requests and then sends the bundle to a server. As this waiting adds delay to responses, the timer should be set properly. For our monitoring application, there is no strict timing requirements, and the timer is set to expire in 2 seconds.

3. Flow Control - in order to prevent the overloading of any server, the proxy monitors the CPU utilization of each server and controls the flow of requests they receive

4. Load-balancing - the requests are distributed among the servers obeying the flow control.

5.1 Architecture

The general architecture of the system is described by figure 5.1.

The components inside the Proxy are different classes in a Object-Oriented architecture used for the actual implementation of the Proxy software, which is described in section 5.6

5.2 Servers

The machines which have the real-time database containing the values of the variables to be monitored run legacy software on top of OpenVms operating system.

The real-time database in this case does not provide an interface for remotely accessing those variables through a TCP/IP network. The only interface natively provided to this database is local, through calls to a C API.

An application was implemented in order to receive requests from remote clients and then issue the local call to the database on the same machine using the C API. This application listens on a given TCP port, through each it serves the requests of the remote clients. It is this server application which we refer to as the "actual server" that receives requests from the proxy/load-balancer.

As the main task of the machine on which our server runs is to serve another higher priority application, we chose to implement this server as simple as possible, consuming very few resources, as a single-threaded TCP server which only translates remote requests to local calls to the C API. Also, we consider the OpenVms operating system to be an environment not very programming-friendly, so we chose to leave all the complexity for the proxy, which runs on Linux.

When the server queries locally the value of a variable through the C API, it makes a call to the database passing the id of the variable as the parameter. In the Oil and Gas industry, this id is more commonly known as "tag".

The local database C API is tag-oriented and only accepts a single tag as parameter. Thus, remote calls to this API would be very inefficient, as many calls would be needed in order to read a great number of variables.

We created a simple wire-protocol for communication between remote clients and the server which is also based on tags but which can handle requests for many tags in a single call, in order to more efficiently use the network resources.

5.3 Wire-Protocol

The Proxy communicates with both the clients and the servers utilizing the same protocol. Thus, clients could communicate directly with the server, but the proxy

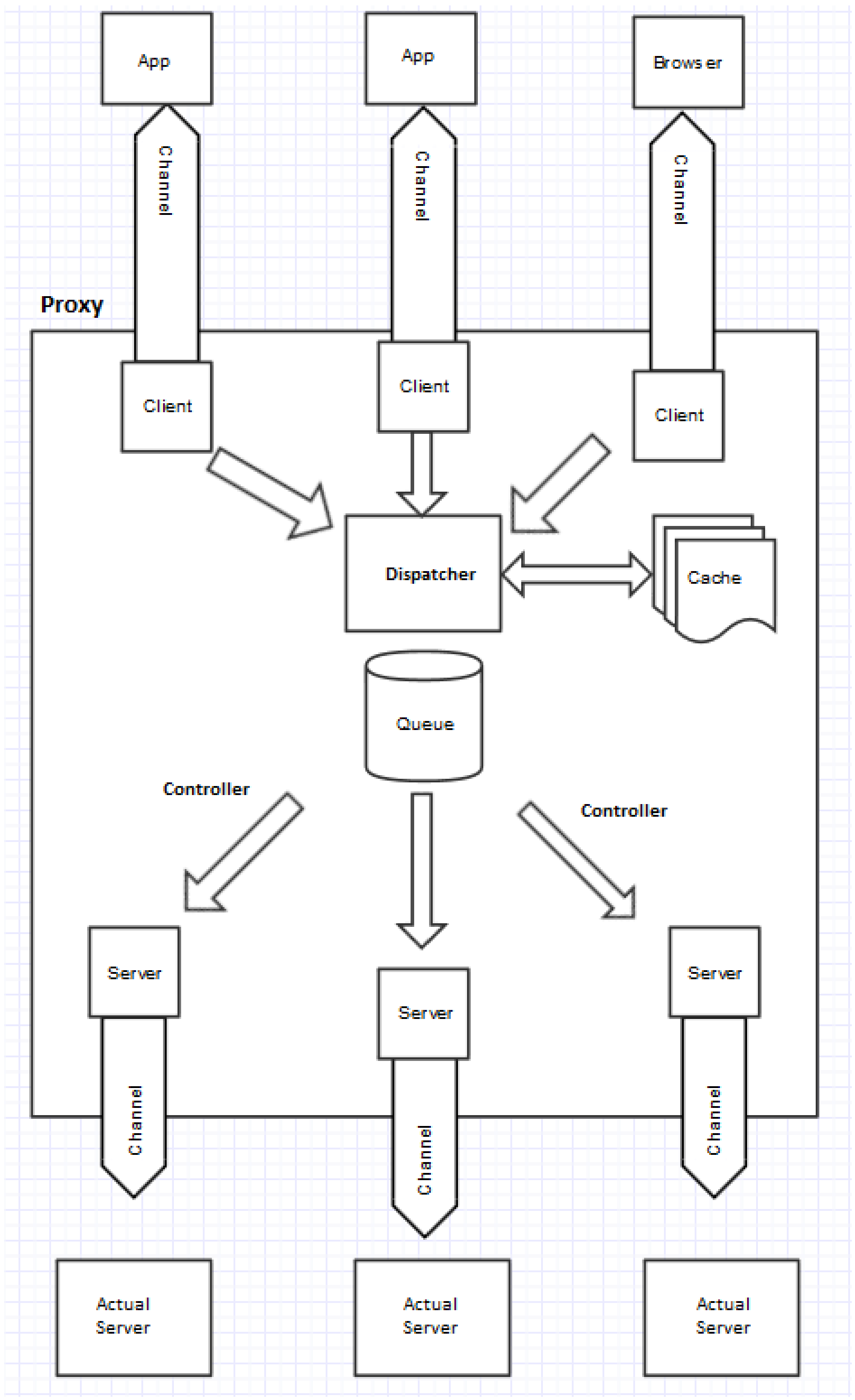


Figure 5.1: Proxy architecture in which the load-balancer is implemented

works as a transparent bridge to provide the aforementioned services - it does not work as gateway, which would make a translation between protocols.

As the servers run a C application on OpenVms, which does not have many available tools, we needed to create the communication solution from a very a low level of abstraction, through the use of Berkeley TCP sockets.

The "wire-protocol" used over the TCP socket is very simple and character-oriented. We created this wire-protocol based on JSON, which is not as bandwidth-efficient as a binary protocol would be, but is more efficient than XML and is also very friendly to use and debug. Moreover, JSON is a very commonly used format nowadays and has many available tools to facilitate implementation.

In summary, the wire-protocol is composed by a request message, which contains a ordered list of the tags being requested, and a response message, which contains a list of responses in the same order as the corresponding requests.

5.4 RESTful interface

As the servers run a C application with very few tools for communication, their interface had to be implemented through TCP sockets. On the other hand, the proxy was implemented on a Linux machine with all the tools available in a productive programming environment. Thus, we also created a different and optional interface in order for the clients to communicate with proxy in a easier way.

A RESTful interface was implemented to enable client applications to easily make requests to the proxy through a simple HTTP GET method - which can be made with a single line of Python code using the "requests" module, for example.

The interface was implemented using PHP and an Apache server which had already been deployed for other uses. The PHP code receives the list of tags requested as parameters of the HTTP GET method and then sends a message containing the same requests to the proxy using the low-level wire-protocol we defined. Thus, this code behaves a server for the RESTful interface but as a client of the proxy.

We used the Apache server for implementing the RESTful interface because it was already being used and had already taken port 80. So, any other way would require clients to change the port of their HTTP request. Moreover, using the Apache server we obtained also other advantages it already provided such as logging, access control etc.

5.5 Clients

The proxy was deployed in a Service-Oriented-Architecture, so that each software concentrates on its own task, providing solutions as services not tightly coupled to others. Thu, services can be mixed to compose different solutions, and an application can behave as a client of a service but as a server of other.

Any application that wants to query the real-time values of the process variables on the remote servers will issue requests to the proxy and then behave as its client.

A browser can query the values of a list of tags and show them for the user, behaving as a client to the RESTful interface, which is provided by the Apache Server. On the other hand, the Apache server behaves as a client of the proxy itself.

A monitoring and trending application queries the RESTful interface periodically to store historical data, behaving as a client. But This kind of client behaves as a server when providing those stored values to other applications.

5.6 Proxy

The proxy was implemented in Python due to its readability and productivity.

An Object-Oriented architecture was used, with a class for each component show in figure 5.1 and summarized below.

Channel handles the low level details of a TCP connection through a socket and does the reading and writing of messages. It is used by Clients and Servers, the classes which communicate with their real counterparts.

Interpreter takes the messages delivered by a Channel and interprets them using the conventions of the wire-protocol we defined. It then transforms those messages into Request or Response objects, which will be handled by the Dispatcher. The Interpreter is a Singleton and it is used by every class which has a Channel connection.

Request is a query for a tag and is created from the interpretation of a request message containing a list of tags, which is split by the Interpreter. The Request objects can then be dispatched in different ways and end up being serviced by different servers. Thus, a request message to a Server may contain Requests from different clients and be different from any of the request messages individually issued by them.

Response is the result of a Request, containing the current value of a tag. It is created from the interpretation of a response message coming from an actual server, which contains the results of queries which were possibly issued by different clients. This list is split into independent Response objects by the Interpreter, which can then be handled and delivered to the correct Clients.

Client is a virtual representation of an actual client. It receives through a Channel a message sent by a client application and uses an Interpreter in order to transform them into Requests. It then handles these Requests to the Dispatcher. When it receives the Responses, it transforms them into a message using the Interpreter and sends it back to the Client through the Channel again.

Cache stores the values for which resulted from the last Request of each tag and its time-stamp. It is queried by the Dispatcher for every new Request. The Dispatcher only puts a new Request in the Queue if the Cache does not have a valid value for it.

Dispatcher receives the Requests which could not be resolved by the Cache and puts them into the Queue, where they will wait to be serviced by one of the Servers.

Queue is used by the Proxy to store the Requests which need servicing. The Servers take Requests from the Queue obeying the rate given by their respective Controllers. The Servers take Requests from the Queue in parallel, and this is the way the load is balanced among them.

Server is the virtual representation of an actual server. It takes Requests from the Queue, transforms them into a message using the Interpreter and sends them through its Channel, doing the opposite when they receive a message from the actual server.

Controller is owned by a Server instance to evaluate how great is the allowed flow of Requests. It implements the integral control algorithm for each Server and can be tuned differently for each one of them. It is an application of the Template Method Pattern.

Monitor is owned by a Controller and used to obtain the feedback measurements of CPU utilization of the respective Server. It is implemented using the Zabbix-Get tool, which reads the data from Zabbix-Agent installed on the same operating system as the actual server. Changing this class will change the feedback metric used by the Controller.

5.7 Communication Pattern

The communication between a Client and the Proxy is synchronous: a Client sends a message containing its Requests and waits until a message containing all Responses is received.

The communication among the Proxy and Servers is asynchronous, using the Queue. The Proxy puts Requests in the Queue and then the Servers, right after delivering the Responses of their last group of Requests, get new Requests from the Queue in a flow rate defined by the Controller.

5.8 Controller

5.8.1 Feedback Metrics

In order to implement the feedback mechanism, a Zabbix Agent was installed on the servers to provide metrics regarding the availability of CPU resources. This feedback measurement solution is simple and Zabbix is a vastly used tool for network monitoring.

We have used classic Linux administration metrics, provided by Zabbix, such as "CPU load" and "CPU idle", for measuring the available CPU resources. The metric "CPU load" measures the average number of process in the scheduler queue and has a correlation with CPU utilization but does not actually measure it, so we did not have good results with that metric. All the results we show illustrate experiments using only the "CPU idle" metric C_{idle}, whose complement, $100\% - C_{idle}$, we used as "CPU utilization".

5.8.2 Integral Wind-Up

One detail worth mentioning regarding the implementation of the integral controller is that, by default, this kind of controller would infinitely increment its control decision N_t if there was no load. This would happen because the controller would try to achieve the CPU utilization target, $C*$, while there is nothing to be processed in order to vary the CPU utilization measurement C_t - a common problem to integral controllers and usually called Integral Wind-up.

To prevent Integral Wind-up, we stop the execution of the control algorithm for a server if a given threshold time interval has elapsed since the last time a request was handled by the load balancer for that server - the threshold was set to three times the period used for sampling and updating the control action, t_s.

Chapter 6

Results and Discussion

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.

In all experiments, we use the fixed constant $k = 0.6$ in 4.5, found experimentally - we started with $k = 1$ and decreased it until there was not overshoot, for guaranteeing the stability of the controller.

6.1 Comparison with Round Robin for Homogeneous Servers

We 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 4.2. 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.

6.1.1 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 6.1 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.

As a result of not existing any feedback mechanism to react to extra high priority load, the client keeps sending requests at the same rate, as shown by figure 6.1.

Figure 6.2 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 thus cannot 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, at time $t = 4$ we 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.

6.1.2 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 surpasses the target when high priority demands arrives to the servers.

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 6.3 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.

Moreover, as the request rate is not being reduced, there is no modification to the client behavior.

As expected, the proposed mechanism behaves quite differently, as shown in Figure 6.3. 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.

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. The Client traffic remains unchanged while load is shifted 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.

6.2 Results for Heterogeneous Servers

Having shown a comparison between our balancing mechanism and the classic Round Robin, we now show how our controller and load balancer perform in an infinite load scenario with heterogeneous servers.

6.2.1 Different CPU Utilization Targets

First, we show the behavior of the controller in a scenario with two equally capable servers which, however, have been configured with different CPU utilization targets.

Figure 6.5 shows that, after the servers start receiving requests, the controller makes sure their CPU utilization stabilize around their respective targets even though they are different. We can also see that the dynamics involved in the control process is the same for both of them, as both CPU utilization metrics stabilize around their target in about 2 minutes.

The controller is adaptive and adjusts its integral factor accordingly, as mentioned in 4, so this behavior was expected, and the experiments confirm that its performance is not dependent on the CPU utilization target and not restricted to any operating point.

After stabilizing, around $t = 4$ minutes, Server 0 receives an extra load which would alone occupy 25% of CPU resources. The controller then reacts and decreases the low priority load sent to that server, bringing the CPU utilization back to the target. Thus, the controller prevents the overloading of the server without ever stopping the low priority requests completely.

In the meanwhile, Server 1 remains unaffected and cannot handle any more requests.

As a result, the time it takes for the Client to have his requests served increases.

6.2.2 Different CPU Processing Power

Now we show the behavior of controllers for two different kinds of servers, one with more processing power than the other - Server 1 has three times the processing power of Server 0.

Figure 6.6 shows that each controller brings its own server to the target CPU utilization, even though the servers have different processing power.

As mentioned in chapter 4, each controller is tuned with the correct parameter t_p for its own server, so that its control action is adjusted to the server's characteristics. In this case, t_p for Server 1 is one third of t_p for Server 0 - meaning three times the processing power. Thus, the controller for Server 1 increases the flow of requests in greater increments, considering a greater fraction of the error $C^* - C_t$ every time it acts.

As we can see, the difference in processing power is then finally reflected by the stabilized flow being about three times greater for Server 1.

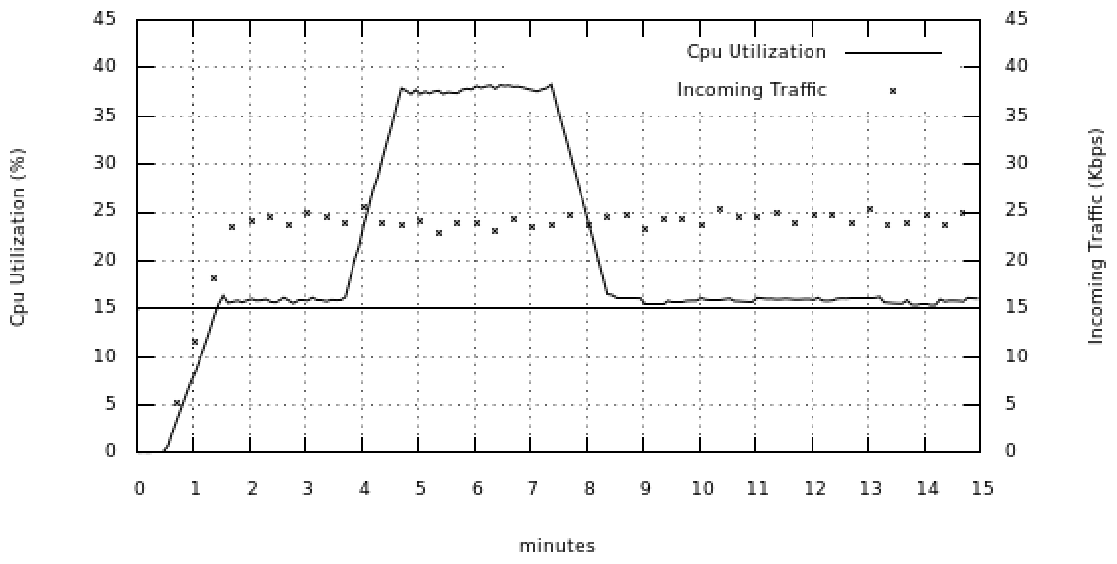
At $t = 4$ minutes, Server 0 receives an extra load which would alone occupy 25% of CPU resources. However, this time the CPU utilization target is set to 15%, so the controller decreases the flow but still cannot reach the target. This makes the flow of requests to Server 0 cease - the measured incoming traffic is background traffic for feedback measurements etc.

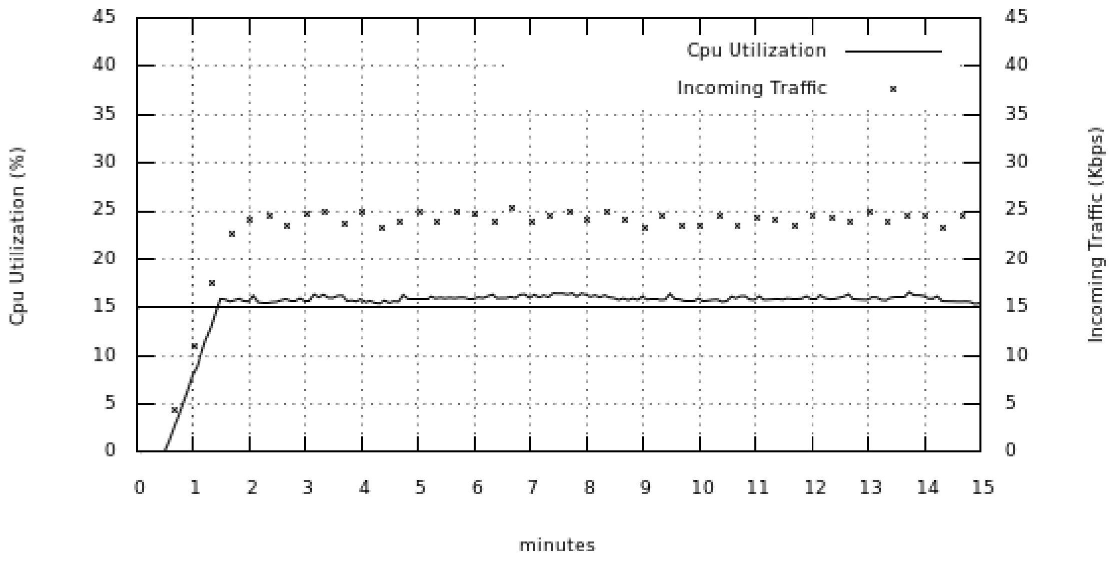
In the meanwhile, Server 1 remains unaffected and cannot handle any more requests.

As a result, in this experiment the behavior of the latency to the Client is different. Instead of the time between client requests increase when one of the servers, Server 0, cannot receive any more flow, it decreases between $t = 7$ and t= 11. This happens because the Client only issues another bundle of requests when it has received the answers for the last one and because the load-balancing mechanism does not take into account the time each request would take to be processed by each server.

When both servers are available, they receive their respective share of requests from a client bundle, and the slowest server, Server 0, becomes a bottleneck, because the Client has to wait for its slow response. On the other hand, when only server 1 is available, it can process all requests by itself, which takes less time due to its greater processing power.

As a consequence, the client graph during this experiment looks like the complement of the the graph for the experiment with different CPU targets.





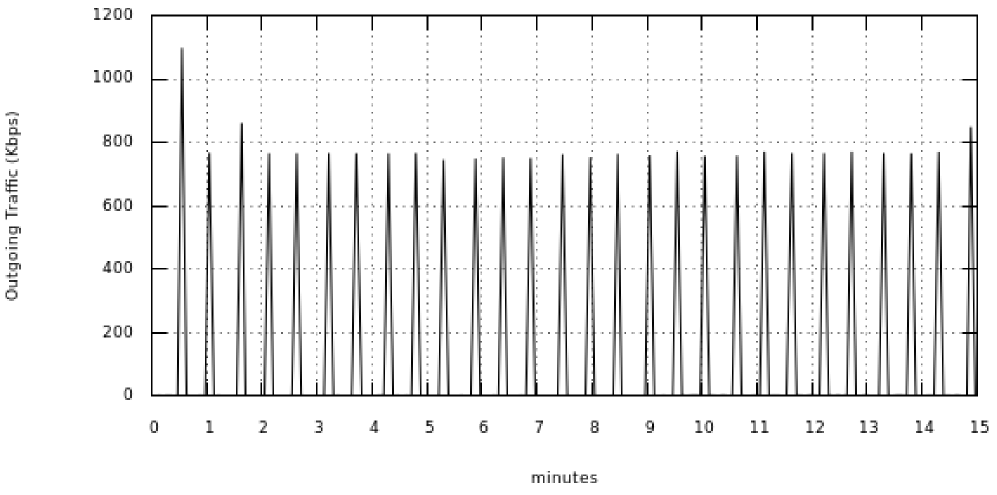
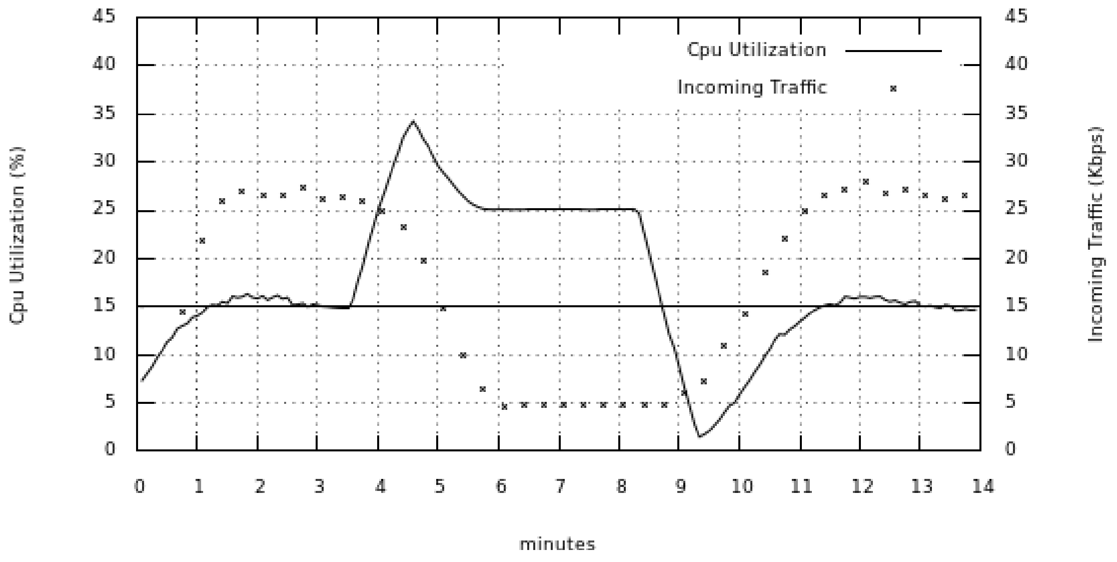
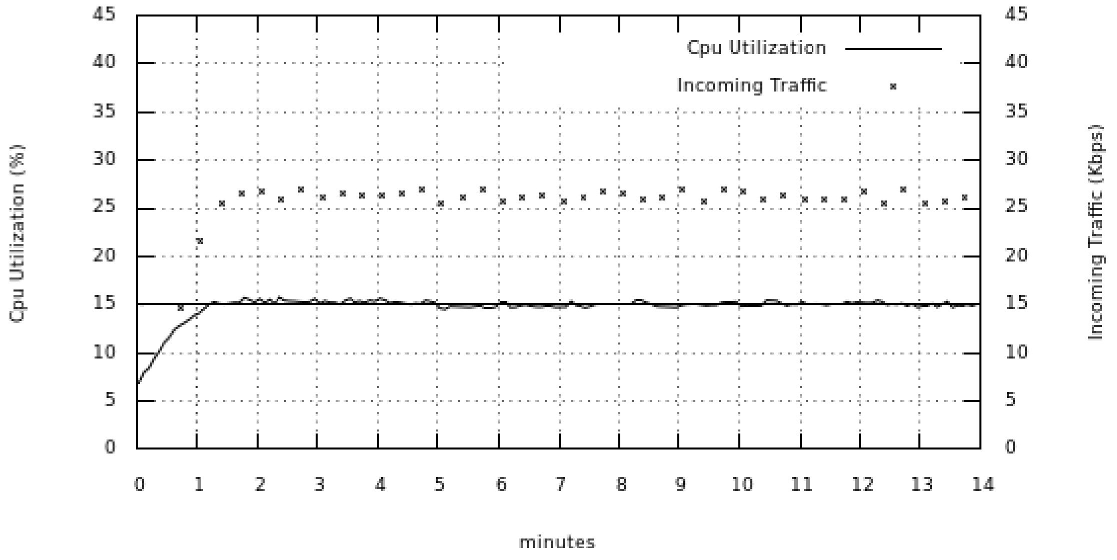


Figure 6.1: Results under modified round robin and infinite load - Server 0 (top), Server 1 (middle) and Client (bottom)





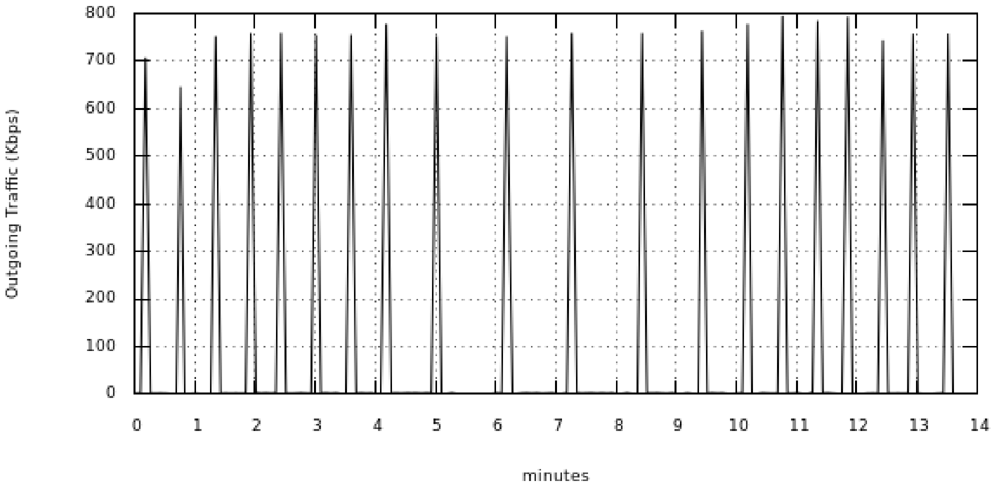
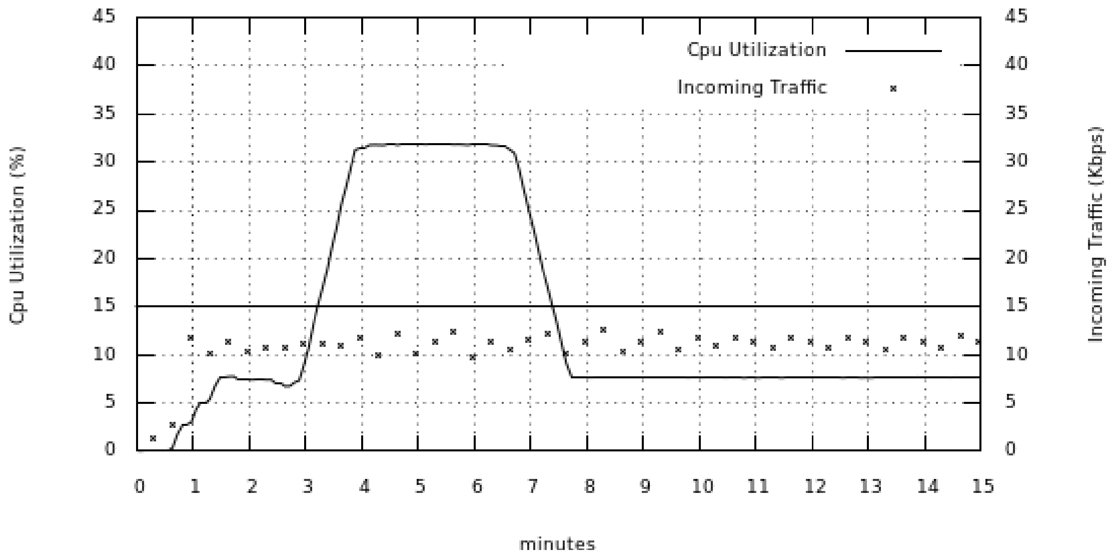
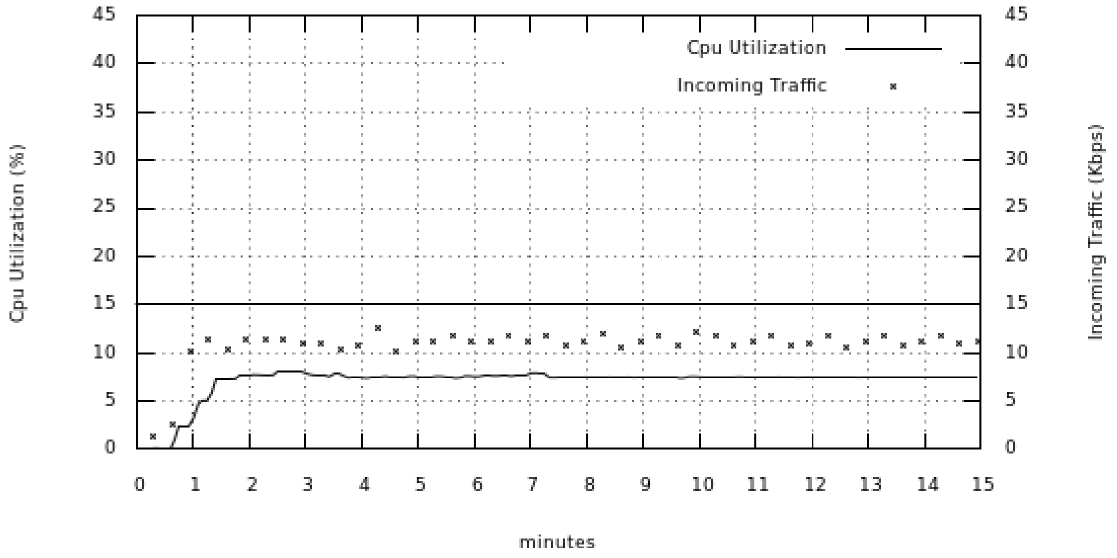


Figure 6.2: Results under proposed controller and infinite load - Server 0 (top), Server 1 (middle) and Client (bottom)





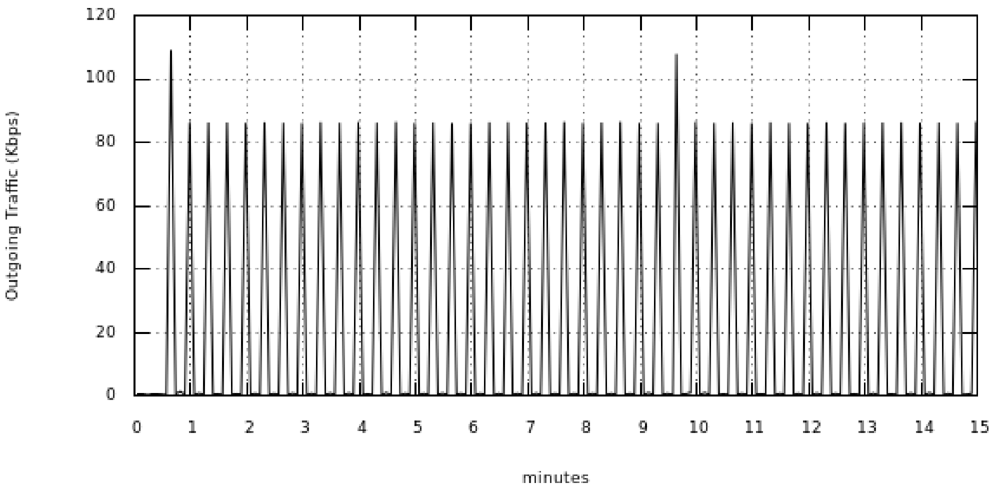
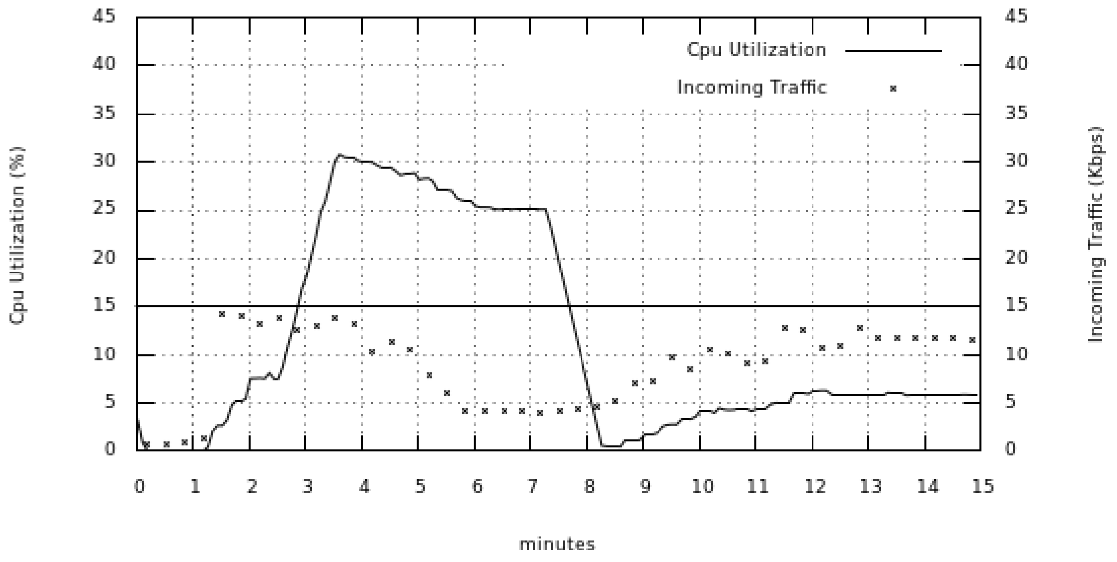
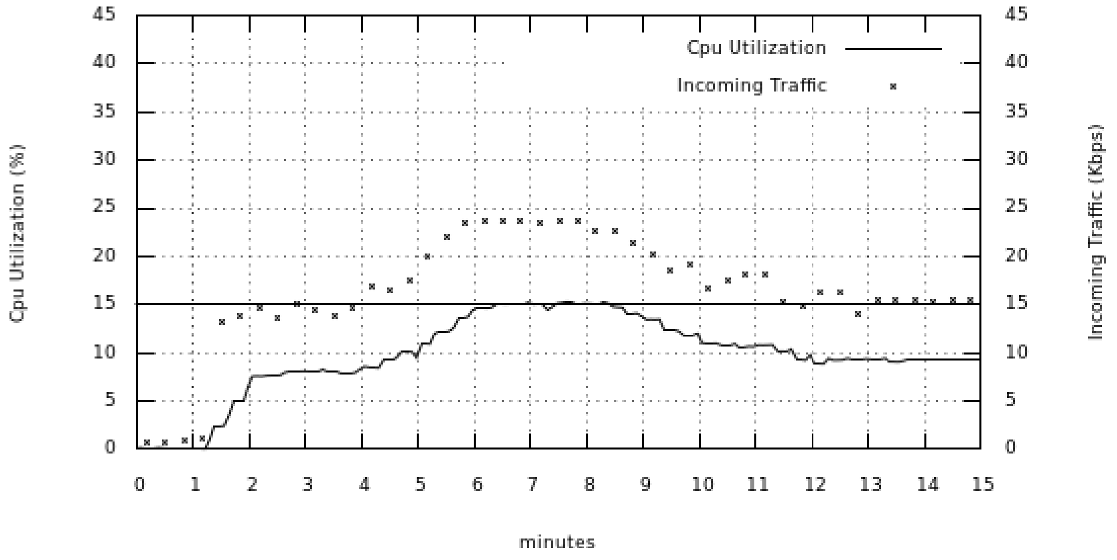


Figure 6.3: Results under modified Round Robin and finite load - Server 0 (top), Server 1 (middle) and Client (bottom)





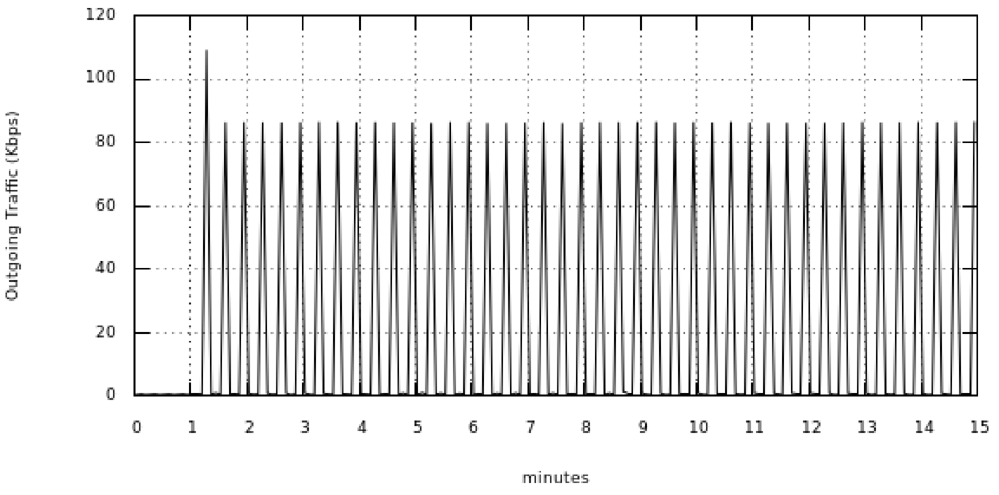
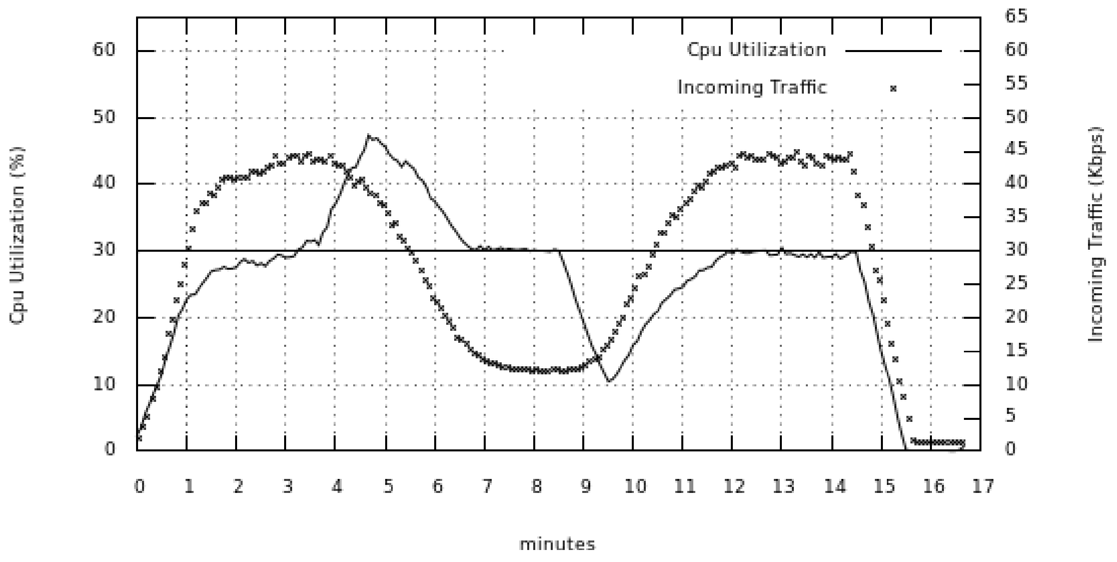
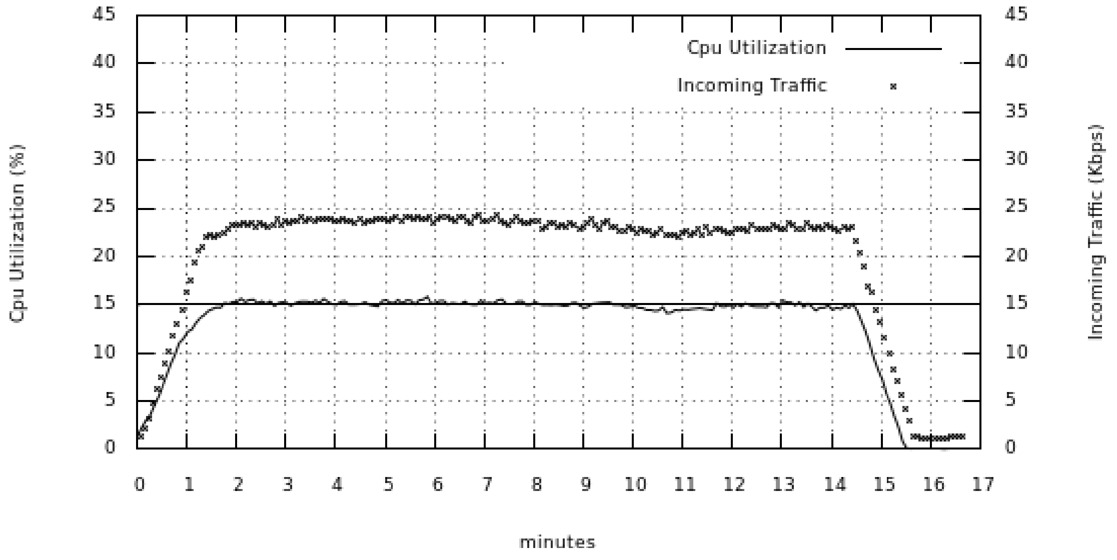


Figure 6.4: Results under proposed controller and finite load - Server 0 (top), Server 1 (middle) and Client (bottom)





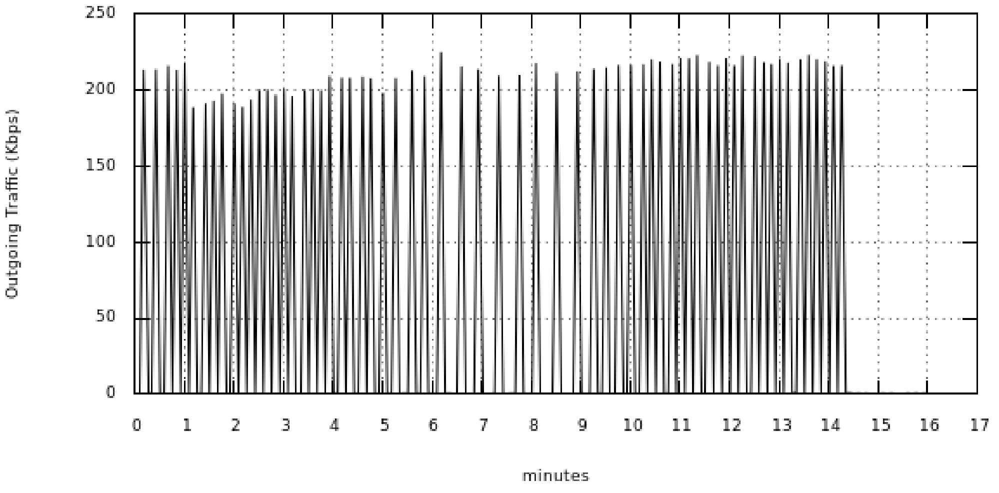
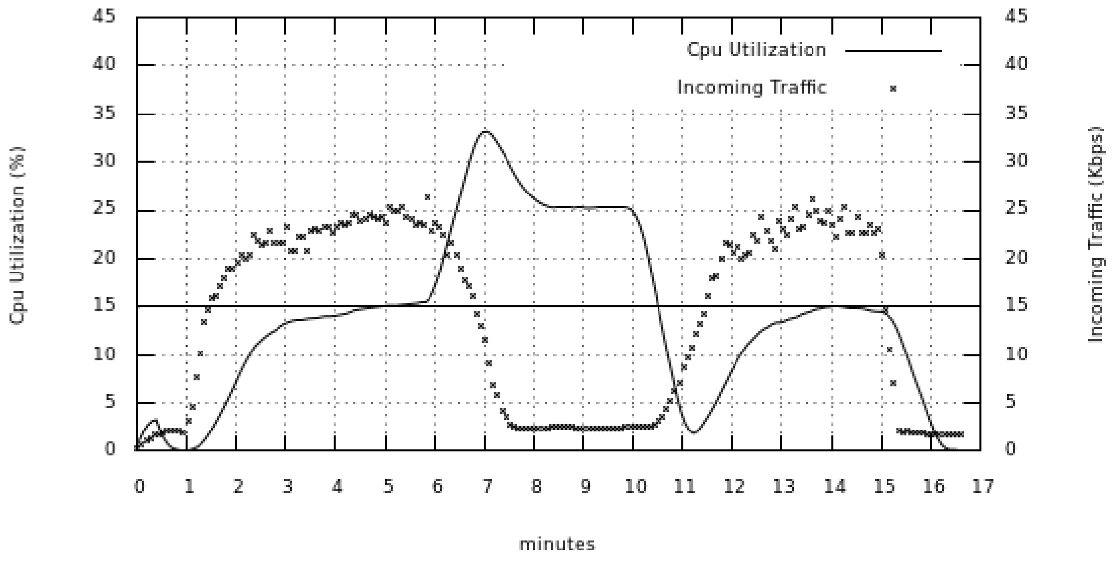
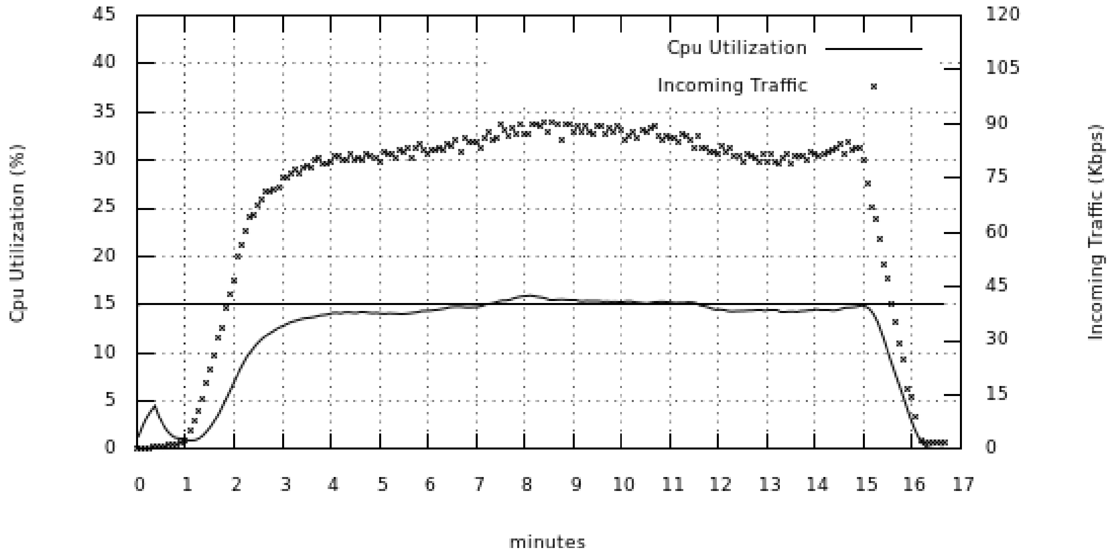


Figure 6.5: Results under proposed controller and different processing targets - Server 0 (top), Server 1 (middle) and Client (bottom)





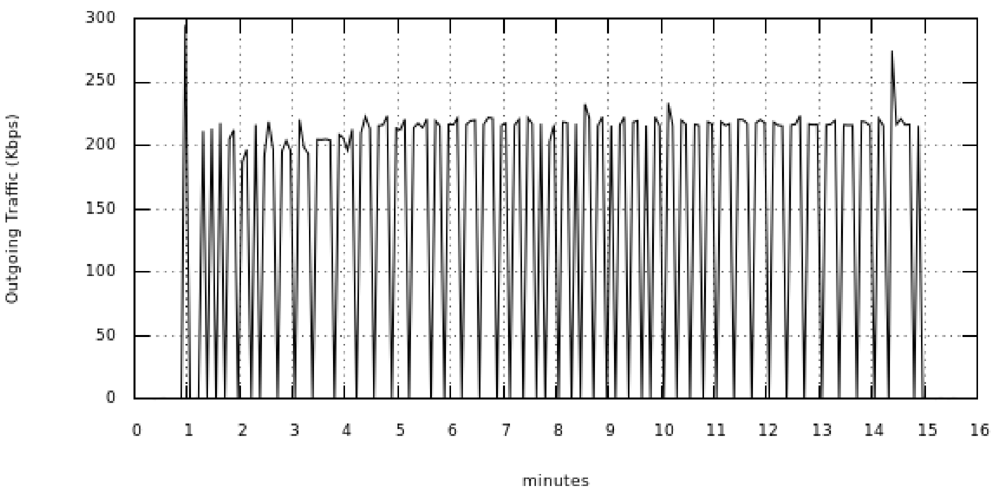


Figure 6.6: Results under proposed controller and different processing power - Server 0 (top), Server 1 (middle) and Client (bottom)

Chapter 7

Conclusions

This dissertation presented a load balancing alternative based on closed-loop controller for scenarios in which servers receive requests from different applications and not all of them is subject to the same load balancer. This kind of scenario prompts for the utilization of feedback measurement for effectively assessing the availability of resources on the servers, and hence a feedback controller is useful.

In particular, in our scenario low priority requests traverse our load balancer while high priority demands can suddenly arrive directly at the servers. So the load balancer keeps CPU utilization arising from low priority requests within a specified target and also distributes the load among the servers.

We propose a simple integral controller and its tuning, based on simple models and intuitive reasoning about the scenario. An integral controller requires only one parameter and is very easy to operate if the tuning needs to be changed. Moreover, it is sufficient for our main goal which is to keep the CPU utilization due to low priority requests within a budget.

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.

The control-based load-balancing algorithm we propose works for servers with different processing power and CPU utilization targets. The results demonstrate how the controllers can operate equally for those heterogeneous scenarios if the proper parameter is tuned.

In the scenario with heterogeneous servers, the balancer does not consider the time a request would take to be processed by a server when deciding which server should receive it. We showed through an experiment that if part of the requests from a client go to a significantly slower server, a bottleneck can arise and the requests may take longer than they would if only the faster server processed them. In that sense, the balancer we propose is not optimal.

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. If it is not possible to measure the variable to be controlled faster and there is a need for faster response times, the same feedback mechanism we proposed could be used more sophisticated control algorithms.

7.1 Future work

The development we have shown along with the results and discussions naturally lead to some improvements and future work:

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

2. Service time prediction: another improvement which can be made is to evaluate the time a request would take to be processed by each server given their current CPU resources availability and consider it when choosing which server should receive it. This would help solve the expected problem illustrated by the last experiment in chapter 6.

3. Priority Queue: an improvement can be made to the queuing policy so that the requests have different priorities. This modification could be used jointly with the modification above, so that higher priorities requests can be serviced faster.

Bibliography

[1] PATEL, P., BANSAL, D., YUAN, L., et al. "Ananta: cloud scale load balancing". In: *ACM SIGCOMM Computer Communication Review*, v. 43, pp. 207–218. ACM, 2013.

[2] GANDHI, R., LIU, H. H., HU, Y. C., et al. "Duet: Cloud scale load balancing with hardware and software", *ACM SIGCOMM Computer Communication Review*, v. 44, n. 4, pp. 27–38, 2015.

[3] LU, Y., XIE, Q., KLIOT, G., et al. "Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services", *Performance Evaluation*, v. 68, n. 11, pp. 1056–1071, 2011.

[4] ALI-ELDIN, A., TORDSSON, J., ELMROTH, E. "An adaptive hybrid elasticity controller for cloud infrastructures". In: *2012 IEEE Network Operations and Management Symposium*, pp. 204–212. IEEE, 2012.

[5] PAPADOPOULOS, A. V., KLEIN, C., MAGGIO, M., et al. "Control-based load-balancing techniques: Analysis and performance evaluation via a randomized optimization approach", *Control Engineering Practice*, v. 52, pp. 24–34, 2016.

[6] DÜRANGO, J., DELLKRANTZ, M., MAGGIO, M., et al. "Control-theoretical load-balancing for cloud applications with brownout". In: *53rd IEEE Conference on Decision and Control*, pp. 5320–5327. IEEE, 2014.

[7] SHAW, S. B., SINGH, A. "A survey on scheduling and load balancing techniques in cloud computing environment". In: *Computer and Communication Technology (ICCCT), 2014 International Conference on*, pp. 87–95. IEEE, 2014.

[8] AL NUAIMI, K., MOHAMED, N., AL NUAIMI, M., et al. "A survey of load balancing in cloud computing: Challenges and algorithms". In: *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*, pp. 137–142. IEEE, 2012.

[9] REN, X., LIN, R., ZOU, H. “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*, pp. 220–224. IEEE, 2011.

[10] DUTREILH, X., MOREAU, A., MALENFANT, J., et al. “From data center resource allocation to control theory and back”. In: *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 410–417. IEEE, 2010.

[11] LIM, H. C., BABU, S., CHASE, J. S. “Automated control for elastic storage”. In: *Proceedings of the 7th international conference on Autonomic computing*, pp. 1–10. ACM, 2010.

[12] LEONTIOU, N., DECHOUNIOTIS, D., DENAZIS, S. “Adaptive admission control of distributed cloud services”. In: *2010 International Conference on Network and Service Management*, pp. 318–321. IEEE, 2010.

[13] XIONG, P., WANG, Z., MALKOWSKI, S., et al. “Economical and robust provisioning of n-tier cloud workloads: A multi-level control approach”. In: *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pp. 571–580. IEEE, 2011.

[14] ASHRAF, A., BYHOLM, B., LEHTINEN, J., et al. “Feedback control algorithms to deploy and scale multiple web applications per virtual machine”. In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 431–438. IEEE, 2012.

[15] AL-SHISHTAWY, A., VLASSOV, V. “Elastman: autonomic elasticity manager for cloud-based key-value stores”. In: *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pp. 115–116. ACM, 2013.

[16] ZHU, Q., AGRAWAL, G. “Resource provisioning with budget constraints for adaptive applications in cloud environments”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 304–307. ACM, 2010.

[17] BEREKMERI, M., SERRANO, D., BOUCHENAK, S., et al. “Feedback Autonomic Provisioning for Guaranteeing Performance in MapReduce Systems”, *IEEE Transactions on Cloud Computing*, 2016.