An algorithm that minimizes cloud computing costs

Marek Szyndler

[1] *Politechnika Opolska, ul. Prószkowska 76, 45-758 Opole , Poland*

Abstract
In the ever-shifting landscape of technology, Cloud Computing emerges as a dynamic force reshaping the very fabric of business operations. As the clouds loom large, businesses find themselves empowered with the flexibility to harness servers, storage, networking, and applications in a way that adapts seamlessly to their evolving needs. In this work is presented a cloud computing model along with corresponding request distribution algorithms responsible for service quality and effective control of the number of activated servers to minimize costs. The presented algorithms use perceptrons to calculate approximate request processing times on servers and waiting times in queues. The first algorithm, based on these times, calculates potential penalties in queues for servers and selects the server for which the increase in this penalty will be the smallest. The next algorithm chooses the least loaded server and inserts a new request into the queue in such a way that the increase in penalty is as small as possible. The last of the presented algorithms aims to distribute requests in a way that maximally diversifies the times remaining for processing requests in the queues.
Keywords
HTTP Request distribution, quality of service, cloud computing, neural networks, perceptrons

1. Introduction

Cloud computing is a service provided by specific software along with the necessary infrastructure. This means the elimination of purchasing licenses, installing, and managing the software by the user, who simply pays for using the service. The operation of cloud computing involves transferring the entire burden of providing IT services to a server with constant access through clients' computers. Therefore, cloud processing is a very convenient solution, saving both time and money for both individual users and businesses, as well as corporations. Currently, the development of cloud computing is progressing rapidly to the point where, in the near future, the entire software, including the operating system, will be moved to a server, so users will only need a client with the appropriate interfaces to communicate with it.

Types of cloud services:

- Infrastructure as a Service – in this model, the provider only offers the infrastructure itself, which includes hardware and network access. The client is responsible for installing the operating system and software and managing them. However, the client does not have control over the cloud infrastructure itself. In this setup, the client pays based on the actual usage of the server's resources (in the case of virtual machines) or for a dedicated server.
- Platform as a Service – in this model, the customer has the ability to deploy both their own applications and compatible ones within the cloud infrastructure. They have full control over these applications but do not control the cloud infrastructure itself, including networks, servers, operating systems, storage, etc. The customer can use these applications themselves or offer them as services. In the case of PaaS (Platform as a Service), the customer is billed based on resource usage, such as CPU time, the number of queries, or data transfer.

Proceedings ITTAP'2023: 3rd International Workshop on Information Technologies: Theoretical and Applied Problems, November 22–24, 2023, Ternopil, Ukraine, Opole, Poland
EMAIL: m.szyndler@po.edu.pl
ORCID: 0009-0008-3441-5761

© 2020 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3628/paper33.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

- Software as a Service – in this model, the customer receives a ready-to-use application running within the cloud infrastructure. The customer does not have control over the cloud infrastructure or the configuration of the application, except for basic settings dedicated to the user. In this case, the customer pays for each use of the application or subscribes for a specified period of usage.

It's worth mentioning that in the early stages of cloud computing development, traditional deployments were common, where multiple applications ran on a single server. This led to resource allocation issues because some of these applications could consume most of the resources, causing delays for others. An attempt to solve this problem was to run applications on separate servers, but this resulted in underutilization of most resources and increased server maintenance costs. The solution to this problem was the introduction of virtual machines. With virtualization, you can run multiple isolated virtual machines on a single physical server, each with allocated resources and its own operating system. Applications are then run on these virtual machines, which, if running on different virtual machines, are completely isolated from each other. This allows for better resource utilization, cost reduction, and improved scalability. In the context of virtual machines, it's also worth mentioning migration, which involves moving a running virtual machine from one physical host to another without interrupting its operation. The next stage was the introduction of containers, which function similarly to virtual machines but with a lower degree of isolation and they share the same operating system.

The goal of this research is to develop a HTTP request distribution system consisting of clusters, servers hosting websites, switches, and clients where all clients will be treated the same in order to minimize the costs of maintaining the entire system under varying workloads. To achieve this there are developed appropriate request distribution algorithms and algorithms for turning servers/virtual machines on and off based on varying system load, what is accomplished by adjusting the number of active clients over time. The operation of the system is simulated using the Omnetpp program [1].

The rest of this paper is organized as follows. Section 2 reviews related work on request distribution and guaranteeing quality of service. Section 3 describes system architecture and developed algorithms. Section 4 summarizes research results and ongoing work.

2. Related work

A crucial element affecting the performance of web hosting service, aside from hardware, is the HTTP request distribution algorithm among virtual machines/servers. Most request distribution algorithms aim for an even distribution of requests among cluster servers, which yields satisfactory results even for simple algorithms such as Round Robin, Weighted Round Robin, and Least Load. The first of these mentioned algorithms assigns incoming requests to consecutive servers. Weighted Round Robin is a Round Robin algorithm that incorporates weights for individual servers (weights can be static or dynamically change with server load). Servers with higher weights are selected more frequently in this algorithm than those with lower weights. A derivative of the Round Robin algorithm is CAP (Client Aware Policy) [2], which distributes requests like Round Robin but within several classes (static requests, dynamically lightly CPU-loaded, dynamically heavily CPU-loaded). The aim here is to ensure that each server is loaded with different types of requests, not just those that heavily load the processor. Another group of algorithms aims to increase the chances of hitting a server's cache memory. An example of such an algorithm is LARD [3], which, for a new request, checks whether the request has been processed by any of the servers. If it has, it directs the request to that server, provided it is not overloaded. If the request has not been processed by any server or if the server that previously handled it is overloaded, the least loaded server is chosen.

More complex algorithms have the capability to adapt to changing working conditions. One of the first such algorithms is AdaptLoad [4], which dynamically adjusts its parameters based on system observations. It relies on the sizes of downloaded documents and is designed specifically for handling static requests. Another example of such an algorithm is FARD (Fuzzy Adaptive Request Distribution) [5]. This algorithm takes into account the request class and the three components of the load on each server in the cluster (CPU, disk, and network card). Based on this information, it calculates the service times for a given request by each server in the cluster. The server with the shortest service time is selected, and after servicing the request, the service times for the specific fuzzy sets used in the

calculation of the selected server's time are modified. A very similar approach to calculating service times is presented in the work [6]. An extension of this algorithm is LFNRD [7], in which the measures of load are the numbers of static and dynamic requests serviced by a given server, and additional parameters of fuzzy sets are modified. In subsequent algorithms, GARDiB and GARD [8,9], the above solution is applied to globally distributed server clusters, both with and without intermediary servers.

Most request distribution algorithms are best-effort algorithms that aim to evenly distribute requests among servers and minimize response time. The need to ensure adequate quality of service has been recognized for some time. In the work of Casalicchio and Cardellini [10], the authors mention the importance of proper request queuing, the use of server clusters in different geographic locations, and making appropriate decisions in request distribution, both at the global and local levels. However, they primarily focus on selecting the right number of servers in a cluster, the appropriate proportions of regular servers to backend servers for handling dynamic requests, and server selection for servicing new requests using the Weighted Round Robin (WRR) algorithm. Another article [11] also focuses on ensuring the desired quality of service. It involves resource reservations by providers within a cluster, followed by redirecting clients to subscriber pages based on resource reservations. The more resources a subscriber reserves, the more clients can access their page. In a different approach presented in the work [12], an algorithm is introduced to minimize energy consumption by cluster servers. This is achieved by turning off idle servers and reactivating them as the system load increases.

However, so far, relatively few request distribution algorithms have considered guaranteed service execution times by servers. Among these algorithms are WEDF, MLF, and GGARDiB [13, 14, 15,16]. The first of these algorithms properly schedules incoming requests to a single server to ensure that the maximum service time for a page is not exceeded. The MLF algorithm, designed for use with multiple servers, not only queues requests but also makes decisions about server selection. It chooses a server with the smallest index for which the calculated service time for the request is less than the remaining time until the request's deadline. If such a server does not exist, it selects the server that can service the request in the shortest time. This algorithm tends to load servers with lower indices more, while servers with higher indices remain less loaded, allowing delayed requests to be directed to them. GGARDiB, on the other hand, initially distributes requests among clusters, selecting the one that can service the request in the shortest time. After selecting a cluster, the server selection process is similar to MLF.

Regarding improving service quality, there are several solutions related to migration. Migration is a resource-intensive and time-consuming process and should not be performed too frequently. In the work [17], a method is developed for migrating individual tasks from one overloaded virtual machine to another less loaded virtual machine instead of migrating the entire virtual machine. In contrast, the work [18] describes an algorithm that reduces the number of migrations by predicting and classifying the loads of virtual machines and appropriately allocating them to physical machines to keep the load on physical machines as stable as possible.

It's worth mentioning that in most of the above-mentioned algorithms, customers are not differentiated, meaning that each customer is treated the same. In production applications, business-criteria-based algorithms are often used to differentiate customers. This typically involves prioritizing customers who pay for the service or allocating more resources to them.

3. System description

For the purposes of this work, it is assumed that the cloud computing system handles HTTP requests and consists of a request distribution switch that routes requests sent by clients between zones. Each zone contains a switch and at least one virtual machine. The switches are responsible for distributing requests between zones or virtual machines/servers. The virtual machines are responsible for processing the requests and then sending them back to the client. Requests follow the same path back that they came from, which is through the switches..

The system consists of a first-level switch which is responsible for load distribution among zones (Fig. 1). Each zone contains a second-level switch, servers along with backup servers, and control modules (Fig. 2). Clients send HTTP requests that are objects of one of several types of web pages (business, health, news, science, sport). When a client sends a request, upon entering the system it first goes to the first-level switch, which sends it to a zone and is responsible for turning servers on and off

in the clusters. Upon reaching the zone, the request goes to the second-level switch, which is responsible for distributing requests among servers to ensure quality of service. This switch also calculates service times for the requests. The second-level switch forwards the request to the appropriate server queue. After leaving the server queue, the request goes to the server, where its processing begins. When the request is completed, it returns to the client using the same path it came from, bypassing the server queue.

A server consists of CPU and disk modules. The request first goes to the CPU module, then to the disk module, and back to the CPU module. Dynamic requests are additionally processed by the backup servers. The number of requests that can be on a server simultaneously is constant and is determined before the simulation begins. The CPU and disk modules are implemented as queues, with service times dependent on the request size.

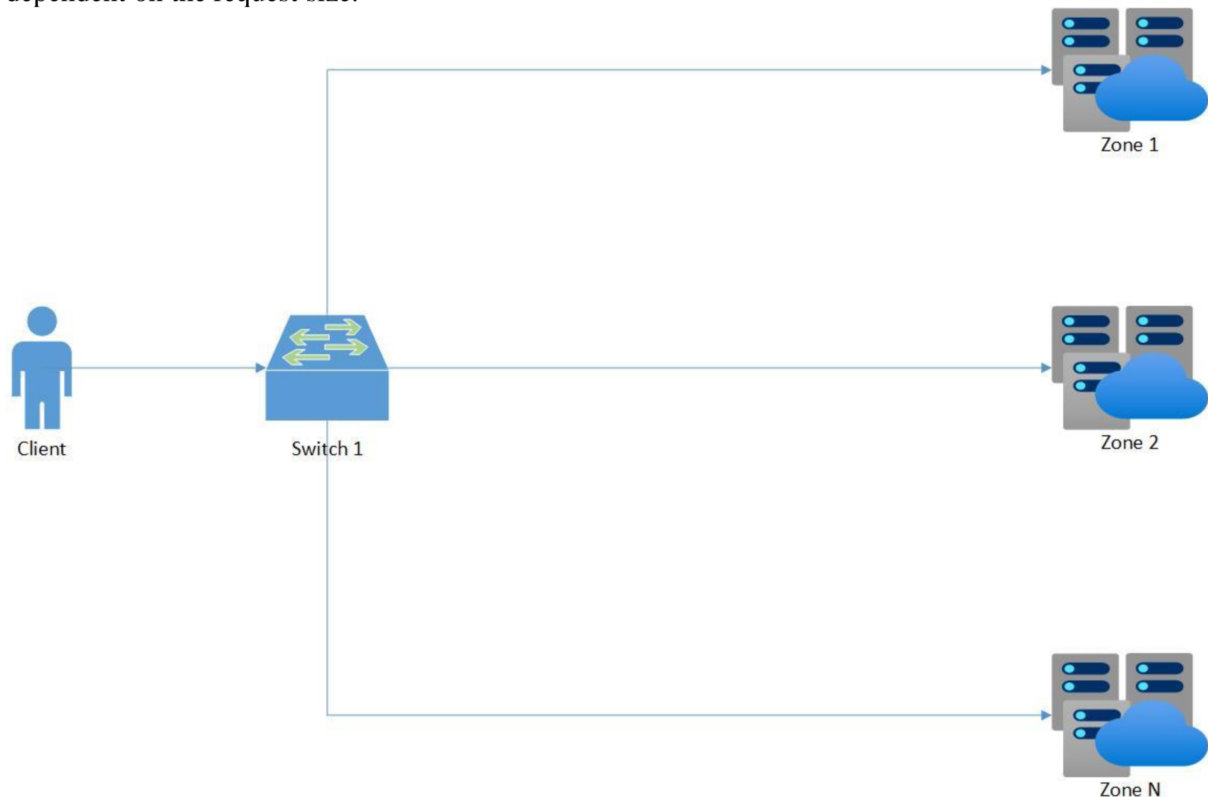

Figure 1: A system diagram - client, first-level switch, and zones.

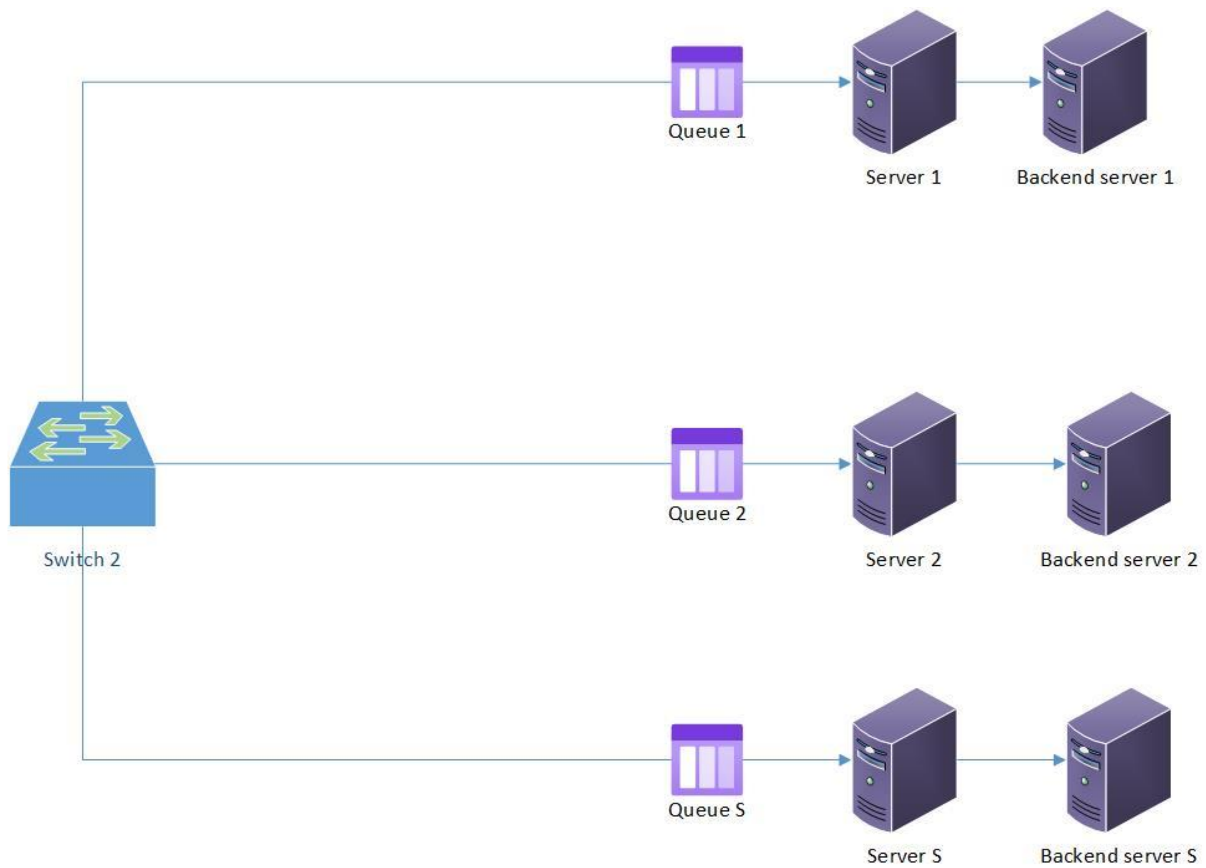

Figure 2: A zone diagram.

3.1. Estimating request service and waiting times

Service time is the time that requests spend on the server. Waiting time is the time that elapses from the entry of a request into the queue to the start of processing on the server. To determine the service and waiting times, perceptrons are used. When a request enters the second-level switch, its size and whether it is a static or dynamic request are retrieved. Based on the size, the request is assigned to the appropriate size class. The input to the neural network consists of the request size divided by the maximum request size in the respective size class, and 0 if the request is static or 1 when it is dynamic. Based on this data, perceptrons calculate the service times for the request on each of the servers.

To calculate the queue time, the input is the appropriately normalized sums of static and dynamic request sizes that are ahead of the given request in the queue. The activation function used in both cases is ReLU (Rectified Linear Unit), $f(x) = \begin{cases} x, for\ x \geq 0 \\ 0, for\ x < 0 \end{cases}$. Figure 3 illustrates a perceptron with n input nodes and 1 layer.

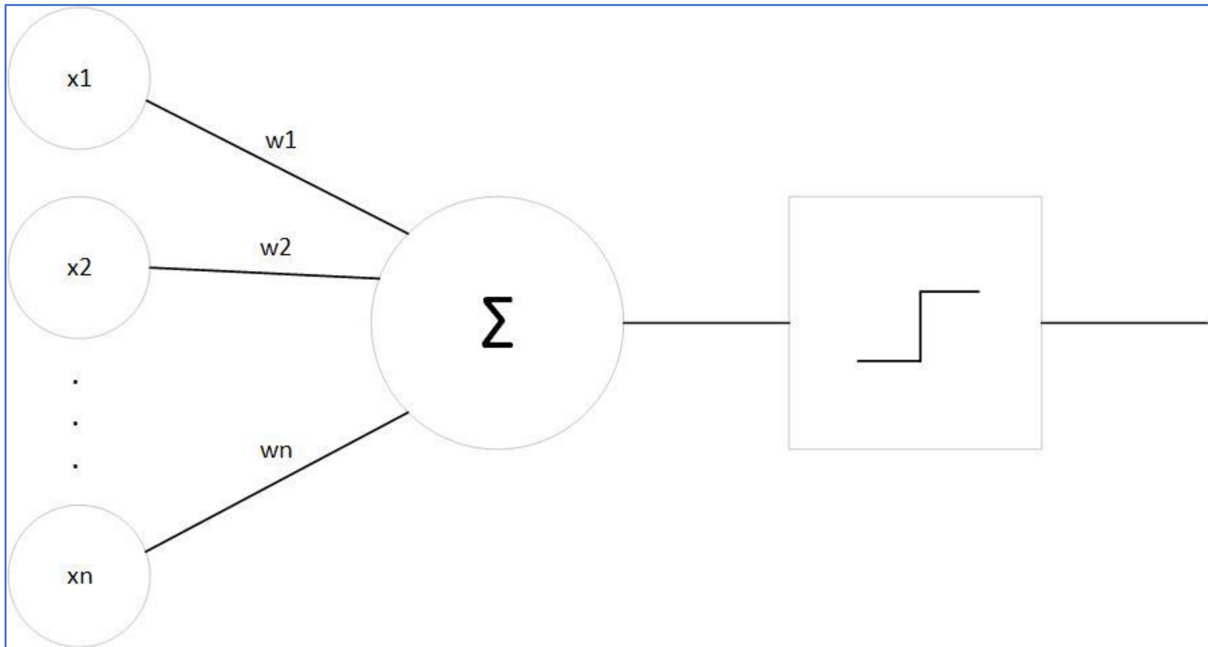

Figure 3: A perceptron with n input nodes.

To update the weights after each request is handled, a function $(f(z)-t)^2$ is minimized, where $z= w_1 x_1 + w_2 x_2$, t is the actual request processing time, or the actual queue waiting time, w_1, w_2 are the current weights, x_1, x_2 are the values of the respective inputs at the moment when request x_i arrives at the second-level switch.

This optimization process aims to adjust the weights w_1 and w_2 so that the predicted response time $f(z)$ is as close as possible to the actual response time t or queue waiting time. The squared difference $(f(z)-t)^2$ serves as the loss function that the optimization algorithm seeks to minimize, thereby improving the accuracy of the model's predictions. Common optimization techniques like gradient descent may be employed to find the optimal weight values.

3.2. First level switch

The first-level switch is composed of the following modules: Request Analysis Module, Cluster Information Gathering Module, Penalty Calculation Module, Decision-Making Module, and Dispatcher Module (Fig 4). The Request Analysis Module collects information about incoming requests, including their type (static or dynamic) and size. The Cluster Information Gathering Module periodically receives information about potential penalties occurring in all queues of servers within a zone. It also maintains data regarding the number of static and dynamic requests in each cluster. The Penalty Calculation Module calculates the actual penalties for delays in requests returning from cluster processing. It also computes the average penalty over time. If this average penalty exceeds the cost of running a server over a given time unit or if the potential total penalty in the queues exceeds a critical threshold, this module decides whether to activate or deactivate another server. The Decision-Making Module determines which cluster should handle a specific request based on algorithms such as Least Load or an algorithm that also considers potential penalties in the queues. It selects the zone where the penalty, per number of requests in cluster is the lowest. The Executor Module is responsible for forwarding requests to the selected cluster for processing. These modules collectively manage the distribution of requests, monitor the state of zones, calculate penalties for delays, and make decisions to optimize server utilization while maintaining quality of service.

Request x_i (where i is the sequential number of incoming requests) arrives at the Request Analysis Module, where its type (static or dynamic) is determined, along with the timestamp $\tau_i^{(1)}$ of its entry into the switch. Subsequently, the request information is forwarded to the Decision-Making Module, which is responsible for assigning the request to a specific cluster. Once the decision is made regarding which

zone will handle the request, the *i-th* request, along with the index of the designated zone, is passed to the Dispatcher Module. This module is responsible for transmitting the request to the chosen zone. Information about the number of requests in each zone is maintained within the Decision-Making Module's array. When a processed request returns from a zone, the index of the cluster it was sent to is retrieved, and the load distribution table in the Decision-Making Module is updated accordingly.

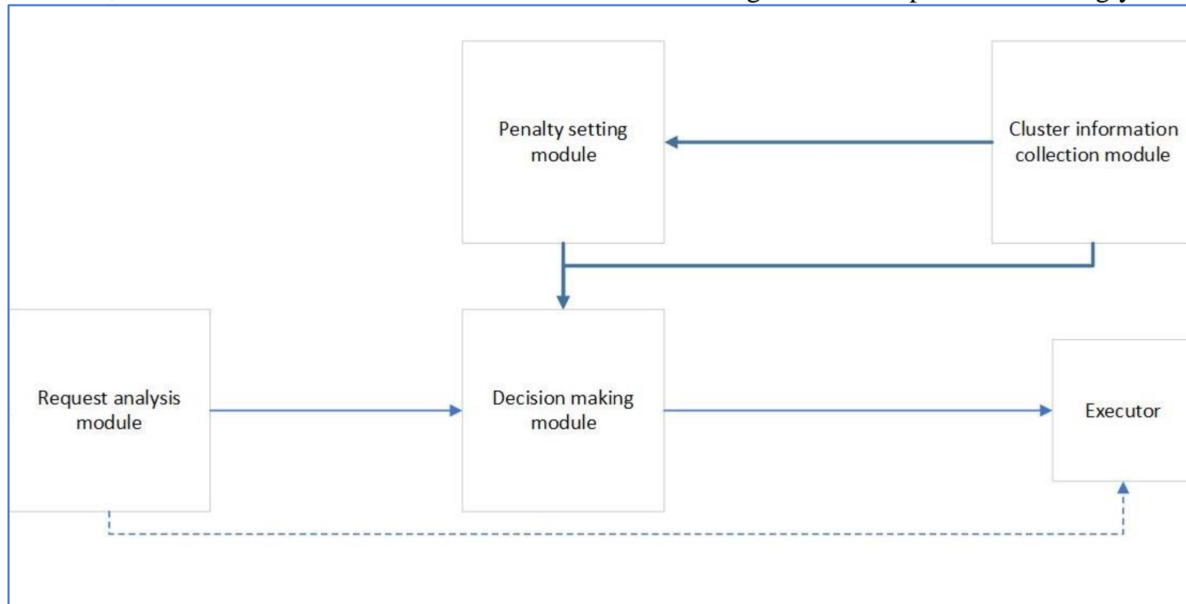

Figure 4: Diagram of a first-level switch. The dashed line represents the path of the request, the solid line conveys information about the request, and the bold solid line carries information about clusters.

3.3. Second level switch with an algorithm that minimizes the increase in penalty.

The diagram of second level switch with algorithm that minimizes the increase in penalty is presented in Figure 5. At the beginning, the *i-th* request, also denoted as x_i, enters the request analysis module where the arrival time $\tau_i^{(2)}$ is recorded. The request type (static or dynamic), size, and delay penalty are determined, and this information is then sent to the service time module, which houses the server models. These models calculate the service times $\hat{t}_i^{(s)}$ for each server in the cluster and the deadlines $d_i^{(s)} = \tau_i^{(1)} + t_{max} - \hat{t}_i^{(s)}$, where $s=1,...,S$, and S is the number of servers in the cluster. t_{max} represents the maximum response time for a request above which a penalty is incurred. The obtained service times, along with other request information, are directed to the penalty module, which receives information from the server information collection module regarding queues and servers. The penalty module then calculates potential penalties for each queue. Penalties are computed both before and after inserting the *i-th* request into the queue. Penalties are calculated for request x_i as well as for requests that are placed behind it in the queue as follows: for a given request in the queue, denoted as x_k, the time the request spends in the queue is calculated. This time is added to the $\tau_i^{(2)}$ timestamp, yielding an estimated time when the processing of request x_k will commence on the server. This estimated time is compared to the deadline d_k for request x_k. If it is later than d_k, the difference between these times is considered a potential delay time for request x_k, and thus, a potential penalty is calculated. Otherwise, the potential penalty is set to 0. In the next step, the array of penalty differences after and before inserting the *i-th* request for each server is passed to the decision-making module. This module selects the server for which the penalty difference is the smallest, i.e., it chooses the index in the array for which the corresponding value is the lowest. Then the request is sent to the queue, where requests are sorted by deadlines *d i.e.* the earlier the deadline *d*, the higher the request is positioned in the queue and leaves it when all requests ahead of it have been sent to the server, and the number of requests being processed by the server is less than the predetermined maximum number of requests per server. After the *i-th*

request returns from being serviced on the server, the actual service time t is sent back to the service time determination module, where the weights for the server model that handled the *i-th* request are updated. Additionally, based on the sum of sizes of static and dynamic requests that were positioned before request x_i, weights responsible for queue time calculation are also updated.

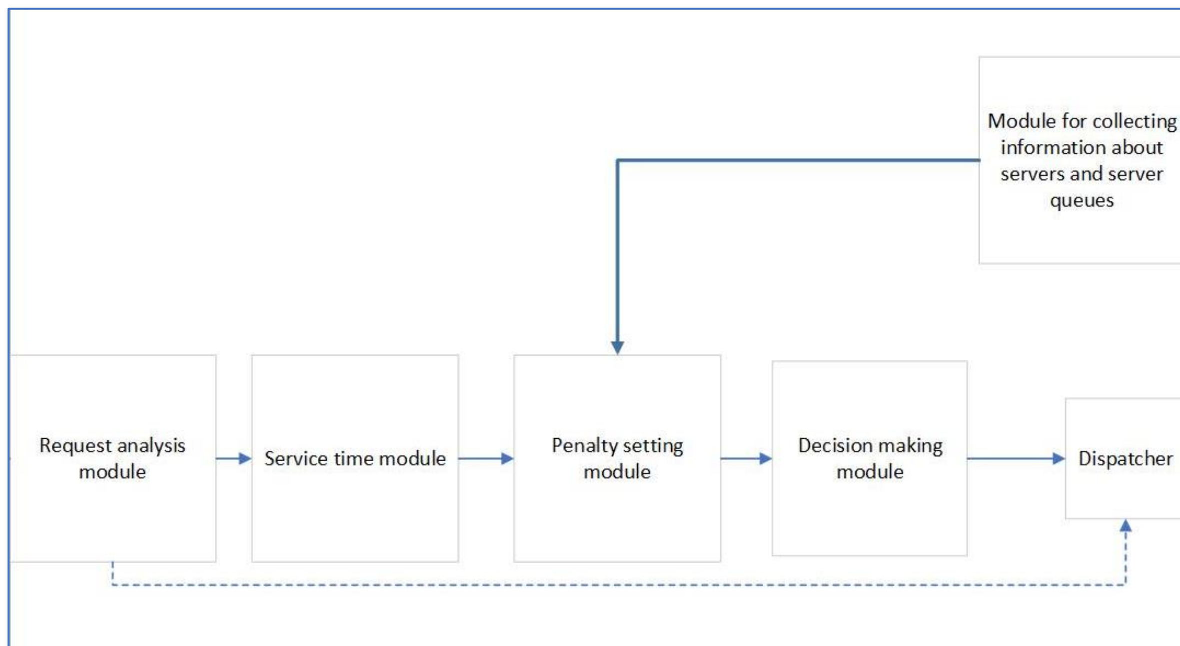

Figure 5: Diagram of a second-level switch. The dashed line represents the path of a request, the solid line conveys information about the request, and the bold solid line carries information about servers and queues for them.

3.4. Second level switch with an algorithm that inserts request into queue so as to decrease the delay penalty

The diagram of second level switch with algorithm that inserts request into queue so as to decrease the delay penalty is presented in Figure 6. Request x_i first enters the request analysis module at time $\tau_i^{(2)}$ and then the service time module, where service times $\hat{t}_i^{(s)}$ for this request on each server are determined, along with the times when the request should begin processing on each server, denoted as $d_i^{(s)} = \tau_i^{(1)} + t_{max} - \hat{t}_i^{(s)}$, where $s=1,\ldots,S$, and S is the number of servers in the cluster. The request then proceeds to the decision-making module, where a server for servicing the request is chosen according to the Least Load algorithm. Subsequently, the service times for the request on the servers, along with other request information, are sent to the penalty determination module. Before the request is placed in the position indicated by the term d_i, an analysis related to request delays and penalties is performed to minimize these penalties (the request may be placed further in the queue than indicated by the term d_i). Finally, the request enters the appropriate server queue where where requests are sorted by deadlines d and leaves it when all requests ahead of it have been sent to the server, and the number of requests being processed by the server is less than the predetermined maximum number of requests per server. The penalty determination module receives periodic updates from both the queue module and the server information module to accurately determine where the *i-th* request should be placed in the queue. The decision-making module receives information from the queue module since, in the case of this algorithm, it selects the server with the shortest queue for this request.

Description of queue penalty calculation:

Before placing request x_i in the queue at the position indicated by the term d_i, we check whether it delays requests that it should precede, starting from the end. Let's assume that there are n requests in the queue for the server, and term d_i points to the *k-th* position in the queue, where $k<n$, and τ_i is the entry time of request x_i into the queue. First, we compare the penalties for delaying request x_i and x_n,

where x_n represents the *n-th* request in the queue, i.e., x_n has a higher position than x_i in the queue, and we calculate the difference in their penalties for subsequent processing: $(\tau_i + \sum_{l=1}^{n} \hat{t}_l - d_i) \times penalty\ for\ 1s\ delay\ x_i$ oraz $(\tau_i + \sum_{l=1}^{n-1} \hat{t}_l + \hat{t}_i - d_n) \times penalty\ for\ 1s\ delay\ x_n$. The symbol $\sum_{l=1}^{n} \hat{t}_l$ represents the waiting time in the queue for a request that is positioned in front of requests with service times $\hat{t}_1$,..., $\hat{t}_n$. The value in bracket indicates the delay, and if it is negative, we assume that the penalty is 0. The first of these values is the penalty for delaying request x_i when it is positioned behind request x_n in the queue, while the second value is the penalty for delaying request x_n when we insert it in front of request x_i. If the second of these values is greater, we insert request x_i behind request x_n; otherwise, we similarly compare the delay costs of requests x_i and x_{n-1}. We continue this procedure until the cost of delaying any request in the queue exceeds the delay of request x_i or when we reach the *k-th* position in the queue.

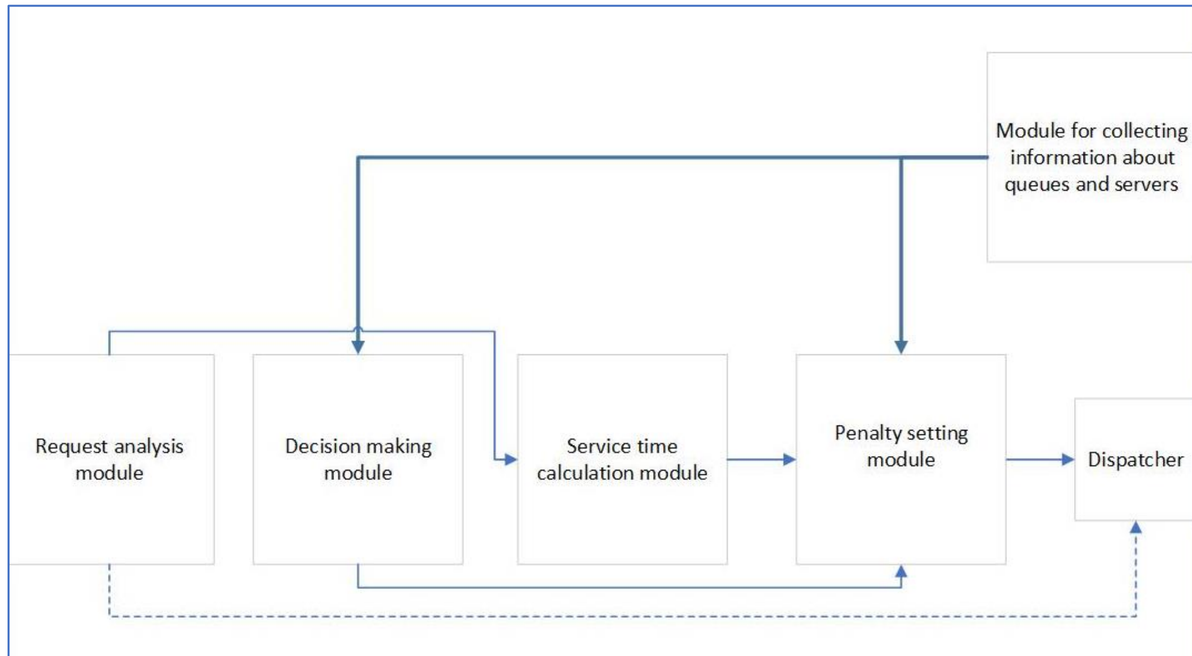

Figure 6: Diagram of a second-level switch. The dashed line represents the path of a request, the solid line conveys information about the request, and the bold solid line carries information about servers and queues for them.

3.5. Second level switch with an algorithm that increases standard deviation of the time remaining to service in queues

The diagram of second level switch with algorithm that that increases standard deviation of the time remaining to service in queues is presented in Figure 7. Request x_i enters the request analysis module, where its size, type (static or dynamic), delay penalty, and the entry time $\tau_i^{(2)}$ into the switch are determined. This information then goes to the service time module, where service times $\hat{t}_i^{(l)}$ and the times when request processing should start on each server, denoted as $d_i^{(l)} = \tau_i^{(1)} + t_{max} - \hat{t}_i^{(l)}$ for $l=1,...,S$, are calculated, S -number of servers in cluster. The calculated service times, along with other request information, are then sent to the decision-making module. This module periodically receives queue state information, especially the terms $d_j^{(l)}$ for requests in the queues, where $l=1,...,S$, and j denotes the index of the request. Based on these terms, times $s_j^{(l)} = d_j^{(l)} - \tau_i^{(2)}$ for $l=1,...,S$, $j=1,...,K_l$, where K_l is the number of requests in the queue for the *l-th* server, are calculated. In this module, a decision is made to choose a server in a way that maximizes the differentiation of times s_j in the queues. For each queue, request x_i is added, and then the standard deviation of times s_j for requests in that queue

is calculated. The standard deviation is divided by the number of requests in the queue for which it is being calculated. The server is chosen for which this value is the highest. If s_j is negative, it is not considered when calculating the standard deviation. The request then goes to the dispatcher module, which sends it to the queue for the selected server. The request leaves the queue when, at a certain point in time τ, an event occurs where the number of requests being processed on the server is less than the predetermined maximum number of requests allowed on the server, and the cost of delaying the remaining requests when selecting that particular request is the smallest. If the queue consists of requests $x_1, \ldots, x_n$ corresponding to their terms, denoted as $d_1, \ldots, d_n$, then when we choose request x_k from the queue for processing, the delay penalty for the remaining requests is equal: $(\tau - d_1) \times \textit{penalty for 1s delay } x_1 + \ldots + (\tau - d_{k-1}) \times \textit{penalty for 1s delay } x_{k-1} + (\tau - d_{k+1}) \times \textit{penalty for 1s delay } x_{k+1} + \ldots + (\tau - d_n) \times \textit{penalty for 1s delay } x_n$. If the expression within any of the brackets is negative, we substitute it with 0, as it indicates that the respective request is not delayed. Each queue has its own penalty module that calculates these costs.

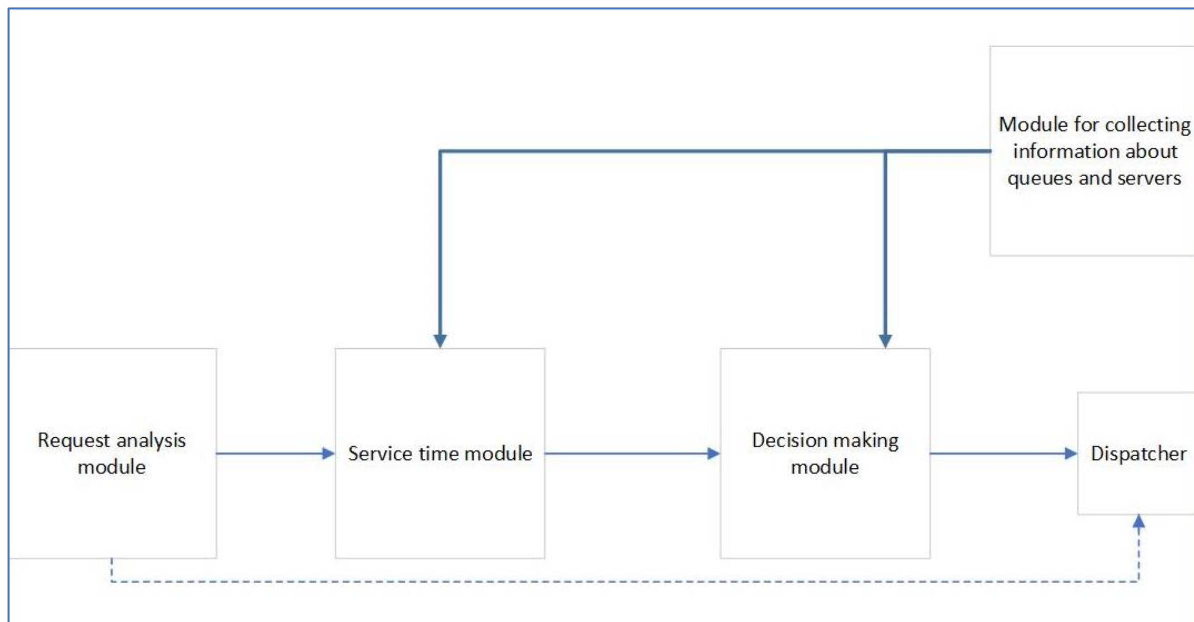

Figure 7: Diagram of a second-level switch. The dashed line represents the path of a request, the solid line conveys information about the request, and the bold solid line carries information about servers and queues for them.

3.6. Control module

The role of control modules is to periodically send control messages. Control Module 1 sends messages to each server in the cluster, where information about the requests residing on the server is recorded. Subsequently, these server-held pieces of information are transmitted to queue modules, where data about the requests within the queues is stored. These messages then proceed to the second-level switch, allowing it to have up-to-date information about the servers and their associated queues. Control Module 2 sends control messages periodically to the second-level switch. These messages contain previously calculated sums of penalties within the queues and on the servers of a given cluster. They are then transmitted to the first-level switch. Each cluster contains one such module.

4. Summary

In this work, are described both the fundamental request distribution algorithms commonly used in production solutions and more sophisticated ones based on fuzzy sets and artificial neural networks. It is also proposed loud computing model along with algorithms to ensure service quality while minimizing costs by maintaining an appropriate number of active servers. The system consists of a first-

level switch responsible for routing requests between clusters and controlling the number of active servers, and, within each cluster, second-level switches distributing requests between servers and control modules responsible for transmitting information about servers, queues, or clusters to the respective switches. To minimize penalties for request delays, there are developed three request distribution algorithms for second-level switches: The first algorithm minimizes the increase in penalties in queues, The second algorithm selects the least loaded server and places the request in such a position in the queue to minimize the penalty, The third algorithm selects servers in a way that maximizes the variation in remaining service times in the queues. These algorithms, along with an appropriate strategy for turning servers on and off, are expected to significantly reduce the operational costs of cloud computing compared to commonly used algorithms like Round Robin or Least Load.

5. References

[1] OMNeT++ https://omnetpp.org/

[2] Emiliano Casalicchio, Michele Colajanni, —"A client-aware dispatching algorithm for web clusters providing multiple services", Proceedings of the 10th international conference on World Wide Web, p.535-544, May 01-05, 2001, Hong Kong, Hongkong

[3] Pai V.S., Aron M., Banga G., Svendsen M., Druschel P., Zwaenpoel W., Nahum E.— „Locality - aware request distribution in cluster based network servers". In Proceedings of 8-th ACM Conference On Architectural Support for Programming Languages and Operating Systems, San Jose, October 1998. SIGOPS Operating System Review, ACM, New York, USA, 1998 vol 32(5), pp. 205-216

[4] A.Riska,Wei Sun, E. Smirni, G.Ciardo — „ADAPTLOAD: effective balancing in clustered web servers under transient load conditions" (ICDCS 2002), Vienna, Austria, pp. 103-111, 2002.

[5] Leszek Borzemski, Krzysztof Zatwamicki. "A fuzzy adaptive request distribution algorithm for cluster-based web systems", In Proceedings of Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, Genova, Italy, February 05 - 07, 2003, pp: 119-126.

[6] Saeed Sharifian, Mohammad K.Akbari and Seyed A.Motamedi, —An Intelligence Layer-7 Switch for Web Server Clusters, 3rd IEEE International Conference on Sciences of Electronic, Technologies of Information and Telecommunications(SETIT 2005), March 27-31, 2005.

[7] Krzysztof Zatwarnicki. „Adaptive Request Distribution in Cluster-Based Web System" Conference: Knowledge-Based and Intelligent Information and Engineering Systems" - 15th International Conference, KES 2011, Kaiserslautern, Germany, September 12-14, 2011, Proceedings, Part I

[8] L. Borzemski, A. Zatwarnicka, K. Zatwarnicki,—"Global adaptive request distribution with broker", In Proc. of 11th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems, Lecture Notes in Computer Science 4693, Springer, pp. 271-278, 2007.

[9] L. Borzemski,K. Zatwarnicki „CDNs with Global Adaptive Request Distribution" Conference: Knowledge-Based Intelligent Information and Engineering Systems, 12th International Conference, KES 2008, Zagreb, Croatia, September 3-5, 2008, Proceedings, Part II

[10] V. Cardellini , E. Casalicchio , M. Colajanni, —"A Performance Study of Distributed Architectures for the Quality of Web Services", Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9, p.9019, January 03-06, 2001

[11] Zhiguang Shan, Chuang Lin, Dan C. Marinescu, Yang Yang, —"Modeling and performance analysis of QoS-aware load balancing of web-server clusters", Computer Networks: The International Journal of Computer and Telecommunications Networking,Vol. 40 No.2, p.235-256, 7 October 2002

[12] K. Rajamani , C. Lefurgy, —"On evaluating request- distribution schemes for saving energy in server clusters", Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software, p.111-122, March 06-08, 2003

[13] Krzysztof Zatwarnicki „Providing Web Service of Established Quality with the Use of HTTP Requests Scheduling Methods" Conference: Agent and Multi-Agent Systems: Technologies and

Applications, 4th KES International Symposium, KES-AMSTA 2010, Gdynia, Poland, June 23-25, 2010, Proceedings. Part I

[14] Krzysztof Zatwarnicki „A cluster-based web system providing guaranteed service" Systems Science 35(4):69-80 2009

[15] Krzysztof Zatwarnicki — „Guaranteeing quality of service in globally distributed web system with brookers Lecture notes in Artifical Intelligence", Springer-Verlag, Berlin-Heidelberg, 2011, vol. 160, pp 45-54

[16] Krzysztof Zatwarnicki, —"Operation of Cluster-Based Web System Guaranteeing Web Page Response Time", In Proceedings of the 5th International Conference on Computational Collective Intelligence, Technologies and Applications, Vol. 8083. Springer-Verlag New York, pp. 477-486, 2013.

[17] Fahimeh Ramezani, Jie Lu, Farooukh Hussain — „A Fuzzy Predictable Load Balancing Approach in Cloud Computing"

[18] Hao Wu, Yuqi Chen, Chi Zang, Jianchao Dong „Loads prediction and consolidation of virtual machines in cloud" 2023