Distribute load among concurrent servers

Denys Bakhtiiarov[1,2,*,†], Bohdan Chumachenko[1,†], Oleksandr Lavrynenko[1,†], Volodymyr Chupryn[1,†] and Veniamin Antonov[1,†]

[1] National Aviation University, 1 Kosmonavta Komarova ave., 03058 Kyiv, Ukraine
[2] State Scientific and Research Institute of Cybersecurity Technologies and Information Protection, 3 Maksym Zaliznyak, 03142 Kyiv, Ukraine

Abstract

A technical implementation option for load balancing among concurrently operating application servers is proposed to mitigate the risks of overload amid substantial unpredictable fluctuations in request flow to the application system and the variable processing durations by each application server. The structural-functional model for load balancing inside the server line of the application system is delineated, and designed to operate under conditions where the incoming request flow from clients is characterized as random, unexpected, non-stationary, and pulsing. A proposal is made for a system that generates a flow of requests to the application server line, ensuring the alignment of the stationary intervals of this flow with the intervals of discrete control for equalizing server load factors. A technological framework for load balancing on application servers is proposed, facilitating the equalization of load factors among application system servers through real-time transmission, allowing the redistribution of a portion of incoming request traffic from more heavily loaded servers to those with lesser loads.

Keywords

request, application, server, client, load balancing

1. Introduction

In practice, when utilizing computerized real-time application systems like 'client/server' that permit remote access for clients via the Internet, such as various interactive help systems, the effectiveness is assessed by the value of τs—the average service duration of each stream of customer requests entering the application system input. A reduced value indicates that the consumer is likely to receive a response to their request more promptly [1]. At low request flow intensities, queues at the application system's input are virtually nonexistent, thereby making τs directly contingent upon the performance of the server hardware hosting the application software. Issues occur when the volume of incoming requests is misaligned with the processing speed of the server infrastructure, leading to the accumulation of unprocessed requests, which in turn results in an unacceptable increase in service request duration and certain instances, the loss of some requests. Given the high intensity of request flow in several applications, it is essential to partition it in real-time into parallel demultiplexed substreams and execute their concurrent online processing utilizing a series of application servers with identical functionality. For instance, as illustrated in Fig. 1. Before the processing of a user's request by an application server, it is initially received by the request redirection server (step 1), which employs a block to ascertain the current application server number designated for the request and allocates the request stream in real-time. Users between the line servers (steps 2 and 3) will implement the distribution strategy outlined below. The request redirection server transmits the IP address of the subsequent application server, as determined by the distribution method, to the user terminal (step 4), and subsequently readies itself to handle a new request from another user, advancing to step 1. The user utilizes the IP address of the designated application server to retrieve the online result of processing his request from that server (step 5). The designated server resolves the application issue and transmits the outcome to the user (step 6) [2].

Specifically, Fig. 1 illustrates that a series of specialized application software and hardware servers process client requests concurrently. Choosing the number of servers in the configuration should align the request traffic intensity with the application system's performance. Nonetheless, the issues get intricate when addressing an erratic and unpredictable influx of requests, characterized by substantial fluctuations in both intensity and duration. In this scenario, due to erratic variations in request volume and the uncertain processing times by application servers, these servers, in the absence of specific interventions, experience uneven and arbitrary loading—resulting in some servers becoming overloaded and consequently losing requests, while others remain underutilized. Unforeseen variations in the volume of requests directed to any application server can impede request processing due to potential transient server overloads.

CPITS-II 2024: Workshop on Cybersecurity Providing in Information and Telecommunication Systems II, October 26, 2024, Kyiv, Ukraine
* Corresponding author.
† These authors contributed equally.
bakhtiiaroff@tks.nau.edu.ua (D. Bakhtiiarov);
bohdan.chumachenko@npp.nau.edu.ua (B. Chumachenko);
oleksandrlavrynenko@tks.nau.edu.ua (O. Lavrynenko);
volodymyr.chupryn@npp.nau.edu.ua (V. Chupryn);
veniamin.antonov@npp.nau.edu.ua (V. Antonov)

0000-0003-3298-4641 (D. Bakhtiiarov);
0000-0002-0354-2206 (B. Chumachenko);
0000-0002-3285-7565 (O. Lavrynenko);
0000-0001-9412-7413 (V. Chupryn);
0000-0003-2244-262X (V. Antonov)

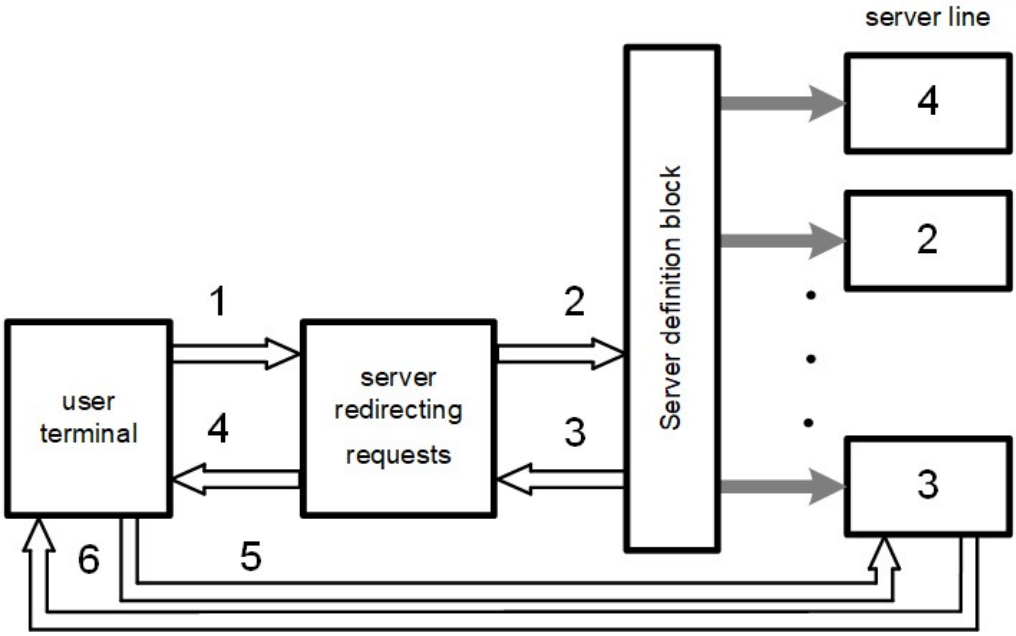

Figure 1: Generalized structural and functional model for the allocation of user requests among application servers

Consequently, there is both theoretical and practical interest in developing a mechanism for load balancing on application servers, specifically a dynamic load balancing approach among collaborating application servers in real-time. This method's implementation aims to avert potential short-term overloads of individual application servers during their operation, thereby fostering the sustainable functioning of the application system amid uncertainties in the dynamics of the aforementioned environmental factors. The suggested technique must assure the stability of the request distribution process, considering the dynamics of unforeseen fluctuations in this flow. The theoretical foundation of this strategy is explained in [3–5]. This paper presents a potential option for its technical implementation, the core of which is as follows. The application system hardware depicted in pic.1 comprises a software server (ROM server+server definition unit) that concurrently and autonomously manages multiple application servers. This software server facilitates a real-time adaptive distribution of requests among the application servers to maintain a more uniform load during unpredictable surges in request flow.

2. Main Part

The theoretical foundation of the employed load balancing method is delineated in [1, 2, 6]. This paper presents a potential option for its technical implementation, the core of which is as follows. The application system comprises a series of application servers that must function concurrently and autonomously, with a software server that facilitates real-time adaptive distribution of request flow among the application servers to achieve more or less uniform load balancing. The parameters of the examined load balancing technology are established through the resolution of the boundary value problem associated with the analytical design of the relevant regulator, utilizing the synthesis of the corresponding R. Bellman functional and iterative numerical integration of the derived tuning equation. The implemented technical solution facilitates nearly uniform loading of server equipment under the specified conditions while maintaining an acceptable average waiting time for service requests with the minimal necessary server resources.

2.1. System model for load balancing on servers

This work introduces a structural and functional model for load balancing throughout the server line of the application system, designed to operate under conditions where the incoming request flow from clients is random, unexpected, non-stationary, and pulsing. Server load balancing entails the real-time redistribution of incoming request flows from heavily loaded application servers to those with lighter loads, thereby achieving a more uniform distribution of load across the servers. Fig. 2 illustrates this model as a series of numbered blocks, each representing a certain functional component of the model's structure [7].

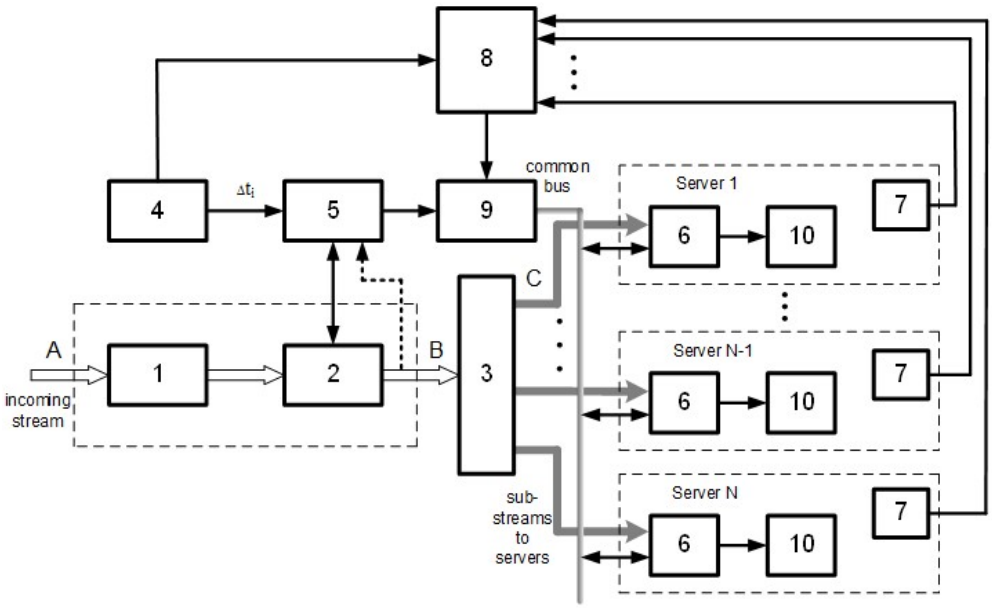

Figure 2: Structural and functional paradigm for load balancing between concurrently operating servers of the applied information system

Fig. 2 use the following designations for functional blocks: 1—smoothing of an input request stream; 2—creation of quasi-stationary segments of incoming request traffic at time intervals *Δti*—smoothing steps (as the formation process is executed as a stepwise iterative procedure with a step *Δti*, while monitoring fluctuations in the intensity of the incoming request flow); 3—demultiplexing of the resulting input stream of requests at each smoothing interval *Δti*; 4—configurator of smoothing and alignment procedures (referring to the process of synchronizing the current values of load factors for application servers seen in pic.2), executed by software-controlled clock generators; 5—assessing the current values of the intensity of the generated input request stream at each smoothing interval *Δti*; 6—buffering requests (establishing a queue of requests for processing by the *i*-th application server) at the input of the *i*-th application server; 7—evaluating the current values of the load factor of the *i*-th application server at each alignment step; 8—determining a singular matrix of regulatory relationships among the variables to be aligned (i.e., between load factors on servers) at each alignment step; 9—ascertaining the precise values of the resource allocation (i.e., the amount of requests) to be allocated among the input queues of application servers at each stage of the alignment; 10—data processing of the relevant issue; *A*—incoming request stream; *B*—produced flow of requests; *B*—query substreams post-demultiplexing. Fig. 2 illustrates that to create quasi-stationary traffic segments, the non-stationary incoming request stream is initially smoothed and structured accordingly. The created input stream is demultiplexed, and the resulting parallel substreams are allocated to the application system's servers based on the established load-balancing method. The primary objective of balancing is to attain the most accurate estimate of the uniform load across the application system servers. In other words, under conditions of unpredictable fluctuations in incoming traffic and varying request processing times by each server, the balancing algorithm must operate to ensure that the generated quasi-stationary traffic segments receive approximately equal load factors across all servers. The model illustrated in Fig. 2 is founded on the adaptive principle of reallocating demultiplexed subflows of requests among application servers through real-time monitoring of fluctuations in the current intensity of the incoming request stream and the existing load levels of the application servers. Consequently, this paradigm necessitates the real-time implementation of the following three processes:

1) The establishment of an incoming request flow to attain a more uniform temporal distribution, thereby preventing short-term overloads in the application server line.
2) The demultiplexing of the incoming request stream into several concurrently operating subflows corresponds to the number of application servers in the line.
3) The equalization of current application server load factors diminishes the likelihood of short-term overload on any individual server. Examine the characteristics of each of these processes.

2.2. Establishment of the incoming request flow

For the proper functioning of this load-balancing method, the incoming request traffic must be transformed into a series of quasi-stationary segments representing a discrete random process, which can be partially refined by specialized averaging techniques. The load balancing technology on the application system's servers necessitates the accurate structuring of request flow, specifically to maintain the consistency between the stationary intervals of this flow, *ΔTs*, and the intervals of the discrete control process for equalizing server load factors, *τk*. Some traffic creation technologies do not allow for this possibility. The "bucket tokens" method [6, 8] has a notable constraint in its applicability, being suitable solely for scenarios where

actual traffic exhibits the traits of a stationary random process. Nevertheless, actual traffic and its derivatives must be regarded as a non-stationary discontinuous process, rendering the straight application of the "token bucket" method, along with other established traffic generating techniques, in adaptive load redistribution systems on servers, largely unjustifiable. This study presents a structural and functional framework for the development of request flow, intended as a component of adaptive load-balancing technology for parallel servers within the application system. This diagram is illustrated in Fig. 3.

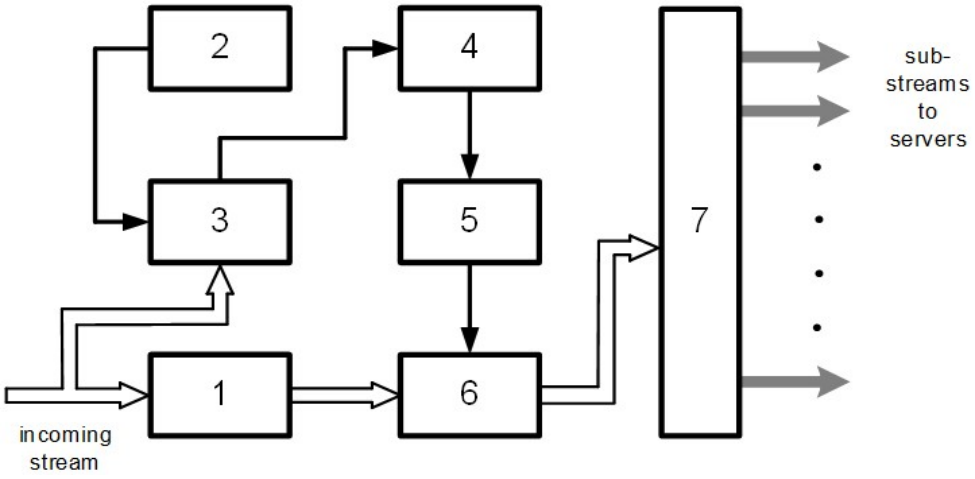

Figure 3: Structural and functional diagram of the request processing pipeline by a series of application servers

Fig. 3 employs the following designations for functional blocks: 1—the request queue buffer at the input of the application system (i.e., the input request storage); 2—the parameter (generator) defining the size of the smoothing step; 3—the measurement of the number of requests received at the input of the balancing system during a single smoothing step duration; 4—generator of virtual events to transmit the request via the gateway (token generator); 5—repository of virtual events for the request sent through the gateway ("bucket of tokens"); 6—gateway for routing requests to the input of the demultiplexer; 7—demultiplexer for the input stream of requests. Fig. 3 illustrates that the foundation of this approach is the 'buckets of tokens' method, but with some adjustments and enhancements that facilitate its application in the processing of non-stationary request flows. In this scenario, the request gateway 6 functions as a lock jumper, allowing requests from the input queue to go to the multiplexer only when the fill level of the 'bucket' of virtual events permits the request to traverse the 'bucket', achieving the average flow rate at the current smoothing step. The velocity of the token generator 4 is contingent upon the strength of the incoming request stream. Based on the intensity measurements conducted by meter 3 at each smoothing step, the configuration of the token generator is executed. Consequently, we acquire quasi-stationary segments of the generated request flow. The applicability of this traffic generation strategy is restricted to instances when there exists a possibility:

1) Establish time intervals, referred to as stationary intervals (ΔTc), during which the average flow rate (Rc) at the input of the load balancing system remains almost constant.
2) Ensure the regulated magnitude of pulsations in the smoothed stream of queries.

The implementation of this traffic processing scheme is warranted if it can transform a non-stationary flow, marked by unpredictable average speeds and fluctuating volumes, into a series of quasi-stationary process segments with defined maximum current thresholds. This transformation enables the implementation of discrete control. The token bucket technique is extensively discussed in the literature, albeit within rather limited domains of applicability. The operational architecture of this algorithm is altered to facilitate its integration into the load-balancing system circuit.

2.3. Demultiplexing the incoming request stream

Demultiplexing the incoming request stream from application system clients is essential when the performance of a single application server is inadequate to effectively process this stream, necessitating the utilization of multiple parallel application servers with identical functionality. One can select from many ways of stream multiplexing. The most straightforward option is to allocate requests from the incoming stream uniformly across application system servers. In this instance, the disparity in request processing times would result in certain servers experiencing temporary overloads, leading to request losses, while other application servers operate under capacity. Consequently, it is prudent to execute the multiplexing of the input stream precisely as seen below.

2.4. Model training

The processing time for each request is an unpredictable variable, resulting in real-time fluctuations of application server load factors. Under these circumstances, balancing server load factors is recommended. Fig. 4 illustrates the structural and functional framework of load balancing on application servers.

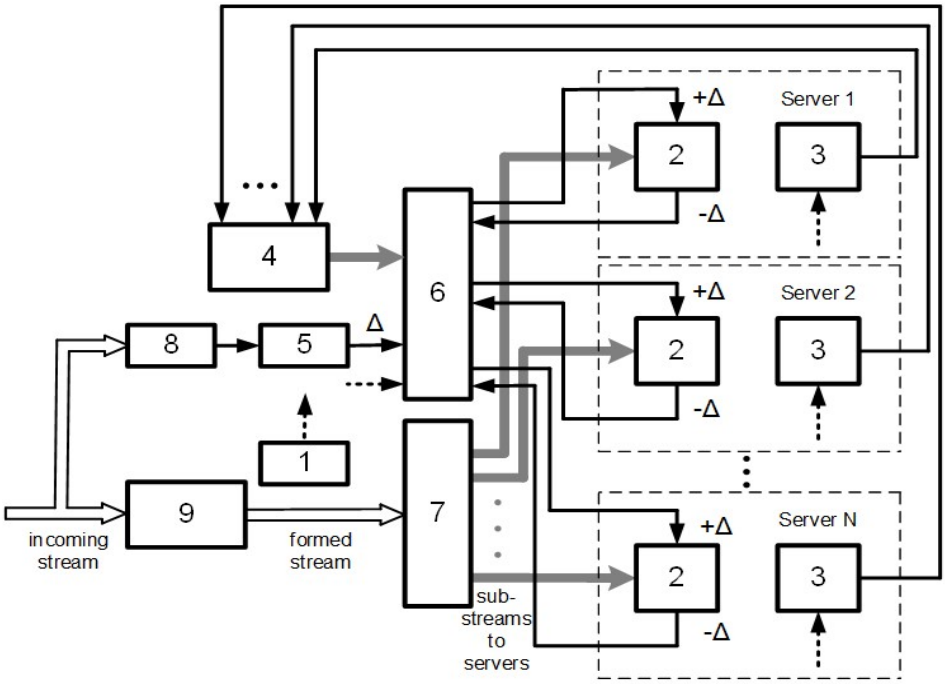

Figure 4: Structural and functional framework for load balancing on application servers

Fig. 4 uses the subsequent designations for functional blocks: 1—settler (generator) of the alignment step magnitude; 2—buffer for the request queue at the server application input; 3—assessment of the current value of the server application load factor (evaluations are conducted at each alignment step); 4—calculation of the determinant of the matrix of regulatory connections among server applications (resulting from the resolution of the configuration equation); 5—computation of the determinant of the resource share Δ (specifically, the number of requests to be redistributed at each alignment step among each server application). The load balancing process is a deliberate iterative procedure for the real-time redistribution of requests inside the request queue buffers for processing at the inputs of each application server. A specific quantity of requests is extracted from one server's queue and subsequently transferred to another server's queue by the established alignment procedure. This redistribution aims to diminish the disparity between the load factor values of the servers comprising the line, facilitating load balancing across each server in the line. The technique operates so that at each alignment step, determined by setter 1 based on the measured current load values of each server, it ascertains the current state of the control link matrix 4 (as a result of the incremental solution). This matrix delineates the direction of request redistribution across server pairs, while the resource share determinant of 5, derived from measurements of current incoming request traffic intensity, specifies the number of requests to be transferred from one server to another. This publication does not include a formal synthesis of the adaptive system controller that executes load balancing on application servers. A synthesis was specifically conducted in [1]. The principles of analytical regulator theory are presented in references [9–14]. Only the subsequent information should be noted. The objective of synthesizing an adaptive controller with a specified quantity of application servers is to mitigate the risk of server equipment overload and to maintain the stability of the load balancing process amidst the unpredictable duration of request processing by each server. The objective of synthesizing such a regulator pertains to the established boundary value problem of analytically designing regulators to minimize the *R*. Bellman functional within the realm of continuous dynamic control systems for entities characterized by ordinary first-order linear differential equations. The application of the synthesis results facilitated a more uniform loading of the server equipment and ensured the requisite stability and length of the balancing procedure despite the aforementioned unanticipated events. The trajectory of traffic flow regulation is dictated by the suitably constructed *R*. Bellman functional. The role of monitoring trends in variations in processed flow intensity on servers is executed through the incremental integration of the relevant differential tuning equation. In the analytical design of the controller, the structure of the Bellman function was defined, enabling the formulation of the tuning equation, the specification of the function, and the derivation of the appropriate Bellman equation. The task of designing a controller is simplified to solving the Riccati equation, a matrix quadratic equation essential for determining the matrix component of the Bellman function. Substituting the identified matrix into the control expression yields the final formulation for the required controller. A regulator is synthesized to maintain a consistent trajectory of state changes in the regulation object's phase space *C2*, adhering to defined quality parameters of the transient process. The controller must observe both the variations in the intensity of incoming request flows and the dynamics of the transient process of load factor equalization to minimize control errors while considering constraints that maintain the stability of the

control system. Initial parameters of the equalization system: the number of servers in the queue and the attenuation coefficient for the Bellman function α. The design of this regulator must address the following inherent physical restrictions. Physical Constraint 1:

$$s_1 + s_2 + s_3 + \ldots + s_n \leq \Delta F . \quad (1)$$

Here's the translated text: where ΔF represents the total bandwidth of the application server line, $\Delta F = f_1 + f_2 + f_3 + \ldots + f_n = const$, $f_1, f_2, f_3, \ldots, f_n$ are the server bandwidths, and $s_1, s_2, s_3, \ldots, s_n$ are the flow intensities of requests at the inputs of application servers.

Physical constraint 2: the unpredictability of request flow ripples.

Physical constraint 3: Ambiguity regarding the processing duration of each specific request by each application server. The efficiency of the load balancing procedure on the servers, from a physical perspective, is the aggregate of the squares of the discrepancies in the load factors of each pair of application servers. This number should be reduced, as a value of zero indicates that the load factors of each server in the line will be identical. Adhering to the aforementioned constraints will decrease the risk of server traffic overflow.

2.5. Essential Factors for Operating PHP Applications Across Multiple Servers

Having addressed load balancing, the subsequent pertinent inquiry is: how are sessions managed? Sessions enable programs to circumvent the stateless characteristic of HTTP and retain information across multiple requests (e.g., authentication status and shopping cart contents). PHP, by default, retains sessions on the server's disk that processes the user's request. For instance, when User A submits a request to Server *B*, a session for User *A* is established and retained on Server *B* (Fig. 5) [11].

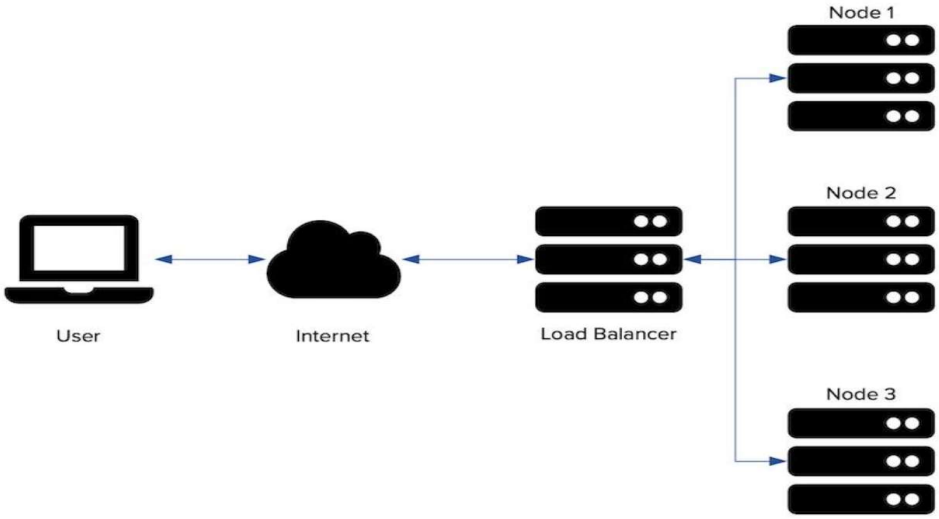

Figure 4: Basic load balancer schematic

Nonetheless, when requests are distributed among numerous servers, this setup is likely to lead to malfunctioning functionality. For instance, consumers may discover their shopping cart is unexpectedly empty midway through the process; they may be arbitrarily redirected to the login page; or they may realize that all their responses in a survey have been erased while completing it. Two alternatives exist to mitigate this: centrally stored sessions and sticky sessions. Centrally Stored Sessions. Sessions may be centrally saved via a caching server (e.g., Redis or Memcached), a database (e.g., MySQL or PostgreSQL), or a shared filesystem (e.g., NFS or GlusterFS). The optimal choice among these choices is a caching server. This is due to two factors: They are an in-memory storage system based on key-value pairs, providing superior responsiveness compared to SQL databases; sessions are consistently written upon the conclusion of a request, whereas SQL databases need writing to the database with each request. This requirement may result in table locking and sluggish write operations. When centrally storing sessions, it is imperative to ensure that the session store does not become a singular point of failure. This can be circumvented by configuring the store in a clustered arrangement. Consequently, if one server in the cluster fails, it is not catastrophic, as another can be incorporated to substitute it [15]. Persistent Sessions. An alternative to session caching is Session Stickiness, also known as Session Persistence. User queries are routed to the same server for the duration of their session. Although it may initially appear to be a wonderful concept, there are various possible downsides, including Will thermal gradients emerge within the cluster? What occurs when a server is inaccessible, overloaded, or requires an upgrade? Consequently, I do not endorse this strategy.

3. Conclusions

In several application systems, such as 'client/server', which exhibit high traffic intensity, the processing of client requests is executed by a series of concurrently operating application servers. Owing to the erratic fluctuations in

request flow and the variable duration of their processing by application servers, these servers, unless specific measures are implemented, experience random and uneven loading—resulting in some servers becoming overloaded and consequently losing requests, while others remain underutilized. In [1], a formal balancing method was developed to avert potential short-term overloads of application servers during their operation, thereby promoting the sustainable functioning of the application system amidst uncertainties in the dynamics of the aforementioned factors. This study presents a potential option for the technical implementation of this strategy.

The structural-functional model of load balancing for the application system's server line is delineated, and designed to operate in conditions where the incoming request flow from clients is random, unexpected, non-stationary, and pulsating. The model utilizes the adaptive principle of reallocating demultiplexed request sub-streams across application servers through real-time monitoring of fluctuations in the incoming request stream intensity and the current load levels of the application servers. This paradigm necessitates the implementation of the following three processes:

1) Establishment of the incoming request flow to prevent short-term server line overloads.
2) Demultiplexing the incoming request stream into multiple parallel substreams based on the number of application servers in the line.
3) Equalization of the current load factor values of application servers.

The formation of an incoming request stream to the application server line is examined. It is demonstrated that the proper functioning of this load-balancing method requires the incoming request traffic to be converted into a sequence of quasi-stationary segments representing a discrete random process. It is essential to align the intervals of stationarity of this request flow with the intervals of the discrete control steps for equalizing the load factor values of application servers. A modification of the established technological approach for packet traffic creation, referred to as the "bucket of tokens", is proposed. The token generator's performance is determined by the intensity of the incoming request stream. Specifically, based on the intensity measurements conducted by the meter at each smoothing step, the token generator is calibrated. Consequently, we acquire quasi-stationary segments of the generated request flow.

A technological technique for load balancing on application servers has been created, characterized as a deliberate iterative procedure for the real-time redistribution of requests stored in the buffers of request queues at the entry points of each application server. This redistribution aims to diminish the disparity between the load factor values of the servers constituting the line. The implemented balancing algorithm enables a specified number of application servers to mitigate the risk of short-term server overloads and ensures the stability of the load-balancing process amidst the unpredictable duration of request processing by each server.

References

[1] D. Bakhtiiarov, G. Konakhovych, O. Lavrynenko, An Approach to Modernization of the Hat and COST 231 Model for Improvement of Electromagnetic Compatibility in Premises for Navigation and Motion Control Equipment, in: 5th International Conference on Methods and Systems of Navigation and Motion Control (MSNMC) (2018) 271–274. doi: 10.1109/MSNMC.2018.8576260.

[2] F. Xia, et al., Community-based Event Dissemination with Optimal Load Balancing, IEEE Trans. Comput. 64(7) (2015) 1857–1869.

[3] A. Nahir, A. Orda, D. Raz, Schedule First Manage Later: Network-Aware Load Balancing, Proc. IEEE INFOCOM (2013) 510–514.

[4] J. Doncel, S. Aalto, U. Ayesta, Economies of Scale in Parallel-Server Systems, Proc. IEEE INFOCOM (2017) 1–9.

[5] O. Veselska, et al., A Wavelet-Based Steganographic Method for Text Hiding in an Audio Signal, Sensors, 22(15) (2022) 5832.

[6] R. Odarchenko, et al., Empirical Wavelet Transform in Speech Signal Compression Problems, in: IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (PIC S&T) (2021) 599–602, doi: 10.1109/PICST54195.2021.9772156.

[7] D. S. Boger, J. S. Fraga, E. Alchieri, Reconfigurable Scalable State Machine Replication, LADC (2016) 1–8.

[8] N. Santos, A. Schiper, Achieving High-Throughput State Machine Replication in Multi-Core Systems, ICDCS (2013).

[9] O. Lavrynenko, et al., Protected Voice Control System of UAV, in: IEEE 5th International Conference Actual Problems of Unmanned Aerial Vehicles Developments (APUAVD) (2019) 295–298. doi: 10.1109/APUAVD-47061.2019.8943926.

[10] O. Solomentsev, et al., A Procedure for Failures Diagnostics of Aviation Radio Equipment, Proceedings—International Conference on Advanced Computer Information Technologies, ACIT (2023) 100–103. doi: 10.1109/ACIT58437.2023.10275337.

[11] D. Bakhtiiarov, et al., Method of Binary Detection of Small Unmanned Aerial Vehicles, in: Cybersecurity Providing in Information and Telecommunication Systems, vol. 3654 (2024) 312–321.

[12] P. J. Marandi, et al., Filo: Con-Solidated Consensus as a Cloud Service, ATC (2016).

[13] M. Poke, T. Hoefler, DARE: High-Performance State Machine Replication on RDMA Networks, HPDC (2015) 107–118.

[14] W. Zhao, Performance Optimization for State Machine Replication based on Application Semantics, J. Syst. Software, 122(C) (2016) 96–109.

[15] J. R. Lorch, et al., Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services, NSDI (2015).