Hindawi
Complexity
Volume 2021, Article ID 5587170, 13 pages
https://doi.org/10.1155/2021/5587170

Research Article

Towards the Concurrent Optimization of the Server: A Case Study on Sport Health Simulation

Nan Jia,[1] **Ruomei Wang**,[2] **Mingliang Li**,[1] **Yuhan Guan**,[2] **and Fan Zhou**[2]

[1]*School of Information Engineering, Intelligent Sensor Network Engineering Research Center of Hebei Province, Hebei GEO University, Shijiazhuang 050031, China*
[2]*School of Computer Science and Engineering, National Engineering Research Center of Digital Life, Sun Yat-sen University, Guangzhou 510006, China*

Correspondence should be addressed to Ruomei Wang; isswrm@mail.sysu.edu.cn

Received 7 January 2021; Revised 24 February 2021; Accepted 3 March 2021; Published 15 March 2021

Academic Editor: Wei Wang

Using computers to conduct human body simulation experiments (e.g., human sport simulation, human physiology simulation, and human clothing simulation) can benefit from both economic and security. However, the human simulation experiment usually requires vast computational resources due to the complex simulation model which combines complicated mathematical and physical principles. As a result, the simulation process is usually time-consuming and simulation efficiency is low. One solution to address the issue of simulation efficiency is to improve the computing performance of the server when the complexity of the simulation model is determined. In this paper, we proposed a concurrent optimization scheme for the server that runs simulation experiments. Specifically, we firstly propose the architecture of the server cluster for the human body simulation, and then we design the concurrent optimization scheme for the server cluster by using Nginx. The experiment results show that the proposed concurrent optimization scheme can make better use of server resources and improve the simulation efficiency in the case of human sport simulation.

1. Introduction

With the development of fundamental subjects (such as mathematics and physics) and computer technology, it becomes possible to use computer technology to conduct the simulation experiments related to human body, such as human sport simulation [1], human physiology simulation [2, 3], and human clothing simulation [4–6]. Human body simulation has many benefits. First of all, the experiment cost is reduced. In the simulation experiment, the experimental objects and equipment are digitized, and the cost is much lower than that of the real experimental objects and equipment [7]. For example, in the human body clothing simulation, the clothing is a digital clothing model, and it is not necessary to design and buy real clothes for the experiment. Secondly, the way of simulation can reduce the risk of the experiment. For instance, in human sport simulation, the simulation scene is used to replace the sport state of real human body in extreme environment to avoid irreversible damage caused by excessive exercise [8].

Although the use of simulation experiments instead of real experiments can bring many benefits, efficient implementation of simulation experiments faces the following challenge. To accurately simulate the dynamic evolution of simulation objects in real environment, the simulation models often build based on complex mathematical modelling as well as following the complicated physical laws. As a result, the simulation model for a specific domain is often very complex. For example, in the process of establishing the heat and moisture simulation model of clothing human body, it is necessary to understand the human physiological tissue structure and construct the simulation model of human thermal physiological regulation mechanism by comprehensively considering dozens of factors such as metabolism, blood flow, respiration, and heart rate [9, 10]. It is also necessary to consider the law of conservation of mass

and energy to construct the heat and moisture transfer model of clothing corresponding to human body and the interaction model between human body and clothing [11]. In order to solve a large number of partial differential equations established in the simulation model, it is necessary to introduce the finite element numerical solution method. The finite element method is famous for its high computational complexity [12], and it takes an average of 16 hours for a single human heat and moisture simulation in our previous study [1].

The complex simulation model means that a lot of computing resources are needed to perform efficient calculations. The traditional single server computing method is obviously not suitable for most simulation scenarios. For example, human sport simulation requires to provide exercise suggestions to users according to the current exercise state, and this puts forward a higher requirement for the simulation efficiency of the model [1]. Usually, when the complexity of the simulation model is determined, the simulation efficiency of the model can be improved by the computing performance of the server. In theory, a large number of high-performance servers can be stacked to improve the computation efficiency of the simulation platform. However, a single high-performance server is often expensive, which is not a good choice for researchers without sufficient budget.

A feasible solution is to use the server cluster technology. That is, multiple servers with lower hardware configuration are gathered together to improve the computation efficiency and provide services for users. Even if the computation efficiency can be improved by server cluster technology, the performance problem of server concurrency needs to be solved. When a large number of users use the simulation platform at the same time, the simulation results will not be obtained in time. In serious cases, the server will crash. Therefore, this paper will focus on how to design and implement a concurrent optimization scheme for simulation server. Specifically, we propose a server cluster method specifically to run the simulation model, in which Nginx (http://nginx.or) is used to build the front-end server as the user access portal, the spring boot framework is used to deploy the simulation model on the back-end application server, and the Redis (https://redis.io/) database is used to save the simulation prediction results. We design and implement a variety of optimization schemes to improve the concurrent processing ability of the cluster, including a dynamic load balancing algorithm combining polling and real-time performance, TCP connection pool, and an efficient thread model of application server. In the case study, we deployed the human sport health simulation model on the server cluster, and the experimental results show that the proposed concurrent optimization scheme achieves a throughput and response time of 70.1 and 3,093 ms when the concurrent number is 500 in the case of human sport simulation.

The rest of this paper is organized as follows. In Section 2 we introduce the basic technology and framework of the server, including Nginx server and load balancing algorithm. In Section 3, we present the designed server cluster architecture as well as the concurrent optimization scheme. Section 4 is the case study, and we implement a sport health simulation platform based on the proposed concurrent optimization schemes and conduct a performance analysis. Section 5 contains conclusions.

2. Background

In this section, we will introduce some concurrent optimization techniques on the server-side used in this study, including NGINX, load balancing technology, and IO multiplexing technology.

2.1. NGINX. Nginx is an open-source high-performance reverse proxy server [13, 14]. In order to deal with the concurrent problem of C10 K (http://www.kegel.com/c10k.html), Nginx designs an asynchronous processing model based on event driven, which can handle a large number of concurrent connections without blocking in a low memory consumption level. At the same time, developers can freely develop third-party modules and integrate them into the Nginx server. The well-known modules in Nginx includes ngx _http_upstream_ fair_module, which implements load balancing based on response time; ngx_nodes_http_healthcheck_module, which implements health check of back-end. At present, Nginx has become the leader in the field of web server, which is adopted by many famous websites such as GitHub (https://github.com/), Pinterest (http://www.pinterest.com), Netflix (https://ir.netflix.net), and Airbnb (https://www.airbnb.cn).

2.1.1. Module Structure. The high stability and scalability of Nginx is closely related to its architecture design. It makes each functional module completely decoupled to eliminate the possible adverse effects between them. The latest version of the Nginx framework defines six types of modules, namely, CORE, EVENT, HTTP, MAIL, STREAM, and CONFIGURATION. The CORE module provides basic functions such as string processing, file reading and writing, digest algorithm, encoding, and decoding. The CORE module also defines some improved data structures based on C language. These data structures can help Nginx process data more efficiently in high concurrency web environment.

The EVENT module is mainly responsible for monitoring connection, adding and deleting read/write events, notifying callback function when the event is ready, and timer and other auxiliary functions. The HTTP module is responsible for receiving, processing, and filtering HTTP requests, including the implementation of reverse proxy and load balancing algorithm. It is the basis of Nginx to provide efficient web services. The MAIL module adds mail protocol to HTTP module. The STREAM module works in the fourth layer of OSI seven layer network model, which can realize the forwarding, proxy, and load balancing of TCP and UDP requests.

The CONFIGURATION module records the configuration information in the whole Nginx framework. Any official or third-party developed Nginx modules must belong

to one of the six categories. The hierarchical relationship among the six modules can be found in Figure 1. The CORE module and CONFIGURATION module are closely related to the Nginx framework. The other four modules are not directly related to the framework, but each has a "manager" in the CORE module.

2.1.2. Process Model. There is usually a master process and multiple worker processes running in the Nginx server. The relationships between them is shown in Figure 2. The master process does not deal with specific web requests. It is only responsible for establishing, binding, closing socket connections, and managing and monitoring the worker process. Worker processes are generated by master process when the server starts. They execute the actual business logic and cooperate with the I/O mechanism provided by the operating system to process HTTP requests and return the response data to the client. After the master process starts, it will suspend waiting for the signal sent by the user. After receiving the signal, it will pass the signal to the worker process through the interprocess communication mechanism provided by the operating system. Each worker process will compete fairly to obtain the request. Then, the master process hangs again, waiting for the next signal. Therefore, the master process is usually relatively idle, while the worker process is busy most of the time.

2.1.3. Reverse Proxy. Reverse proxy [15] is a core function of Nginx. Nginx generally plays a front-end server of the cluster to receive client requests, then forwards the request to the upstream server (that is, the server in the cluster that processes the business logic), and finally receives the response data and returns it to the users. When the number of upstream servers is more than one, the load balancing algorithm is needed to select the forwarding target server. The client thinks it is communicating directly with the reverse proxy server, and it does not know that the request has been processed by the upstream server.

Nginx's reverse proxy method shortens the time for the upstream server to maintain the connection. At the same time, because of the internal LAN communication between the proxy server and the upstream server, the speed is faster, so it can effectively reduce the concurrency pressure of the upstream server.

2.2. Load Balancing Technology. Load balancing [16] technology is usually used together with server cluster. Its idea is that multiple servers with equivalent status form a set, and the request is allocated to a server in the cluster through some algorithm, and the server receiving the request independently responds to the request of the client.

There are three main ways to achieve load balancing [17–19]. The first is based on DNS domain name resolution. The specific method is to bind multiple IP addresses for domain name in DNS server. In this way, the IP address corresponding to the domain name obtained by each client is different, and the request is sent to different servers. The second is based on NAT (network address translation) technology. A NAT server is used as the only channel between the external network and the internal LAN of the cluster. When the NAT server receives an external request, it modifies the target IP address in the request message to redirect to a machine in the cluster. The third is based on reverse proxy: the proxy server acts as the front-end server of the cluster. When the user's request arrives at the proxy server, the proxy server selects a back-end server, rewrites the request, and initiates access to it. The load balancing of Nginx is based on reverse proxy [20, 21].

In the initial stage, static performance of each back-end server is calculated according to its CPU configuration, as shown in equation (1) where performance $[i]$, frequency $[i]$, and cores $[i]$ represent the static performance, CPU main frequency, and CPU core of the i_{th} back-end server, respectively. We then set the static weight sw for each back-end server based on the static performance, as shown in equation (2), where $\text{performance}_{\text{Min}}$ represents the minimum static performance for all back-ends servers. Thus, the server with the worst static performance must have a static weight of 10:

$$\text{performance}[i] = \text{frequency}[i] * \text{cores}[i], \tag{1}$$

$$sw[i] = \frac{\text{performance}[i]}{\text{performance}_{\text{min}}} * 10. \tag{2}$$

2.3. IO Multiplexing. IO multiplexing technology can monitor the status of multiple sockets in one thread at the same time. epoll [22] is the best IO multiplexing method in the Linux system. Before epoll was born, select and poll [22] are the most commonly used IO multiplexing methods in the Linux system. Their characteristics are as follows: when collecting events, they will pass all sockets of established connection to the operating system kernel, and then the kernel will find whether there are unprocessed events on these connections. The disadvantage of this method is that in the high concurrency scenario, the server process maintains connection with a large number of users at the same time, and a large number of sockets are passed to the kernel every time the event is collected, resulting in a large number of memory copies from the user mode to the kernel mode. Therefore, its efficiency is inversely proportional to the number of events monitored. This is particularly wasteful when the server maintains a large number of connections, but only a few connections have events. For example, only 200 out of 100 000 connections actually have events to handle. In order to process the 200 connection requests, the server needs to check all 100000 connections.

However, epoll has different processing mode, and it separates the two parts of sending socket to kernel and getting event and divides the original call of select and poll into three parts: calling epoll_create to create an epoll object and calling epoll_ctl to add connected socket to epoll object, and calling epoll_wait to collect the connection where the event occurred. epoll is efficient because it can add connections to the event_poll object in the kernel to be monitored in advance, so that every time an event is collected,

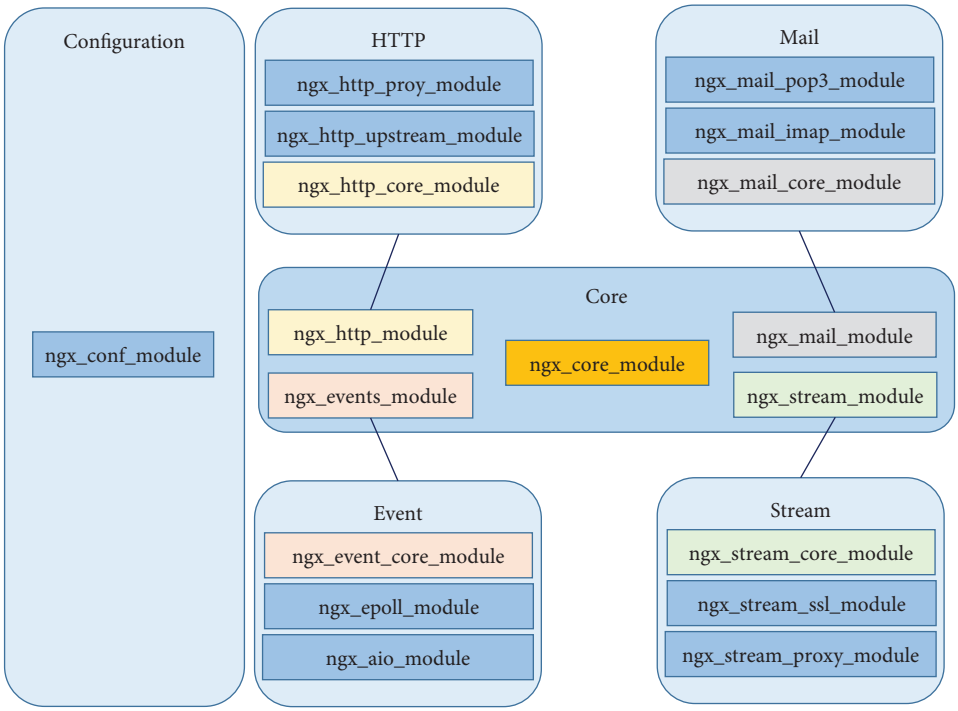


Figure 1: Relationship between Nginx modules.

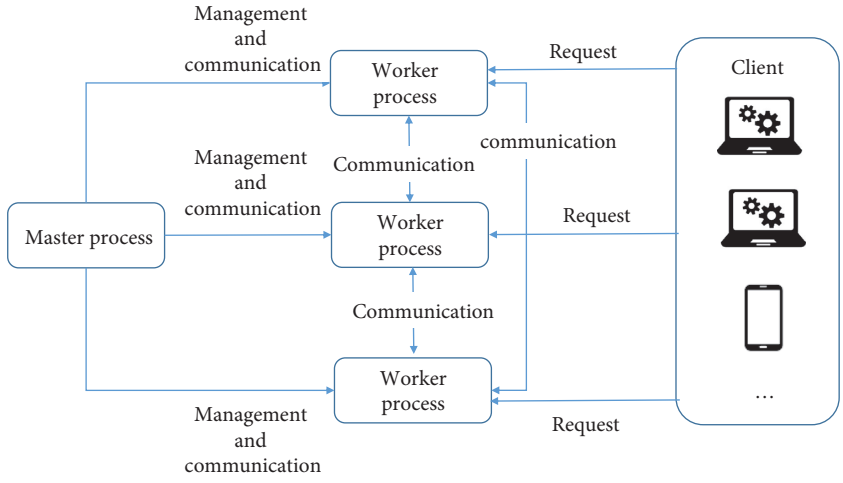


Figure 2: Relationships between master process and worker processes.

there is no need to pass all connections to the kernel, and the kernel does not need to traverse and check all connections. Therefore, epoll can handle a large number of connections more efficiently than select and poll and has more advantages in concurrent scenarios.

3. Sever Concurrent Optimization Scheme

In this section, we firstly introduce the architecture of the server cluster we proposed for the human body simulation, and then we will design the concurrent optimization scheme for the server cluster.

3.1. Design of Server Cluster Architecture. The traditional server architecture is a single-thread model in which user's requests follow a first-come-first-served rule. This architecture is not problematic in situations where user's requests are low or arrival times are scattered. However, when the number of user's requests increases or is concentrated over a short period of time, that is, when there are high concurrent requests, the waiting queue can be long. In the human body simulation scene, the thread handling each request should processes a time-consuming simulation calculation. These time-consuming processes are accumulated, resulting in an increase in the average user wait response time.

In order to reduce the average waiting time of users, we can analyse and design from reducing the actual processing time and reducing the queuing time. However, the human simulation models usually include most iterative operations, and it is difficult to divide into parallel tasks. Therefore, the actual processing time cannot be reduced.

To reduce the queuing time, we can replace the single-threaded model with a multithreaded model. In the multithreaded model, the server starts a new thread for processing each request it receives without waiting for the previous one to complete. Since each thread consumes a certain amount of memory resources, this approach consumes space as well as time. Therefore, we design the server thread pool to handle user requests. The thread pool will maintain in an interval number of threads and avoid running out of memory. At the same time, the thread pool can maintain a waiting queue to store the requests that are failed to deal with at the first time. Because the threads processing the requests are more than one, the waiting queue length will be shorter than using a single-thread model.

To further improve the ability of the server handling concurrent requests, we use a cluster of multiple servers instead of a single server. User requests are distributed reasonably to each machine to achieve load balance. Figure 3 shows the cluster architecture. A front-end server is set up in the cluster. Its IP address is exposed to the public and serves as the unique entry port for users to access the server. The front-end server is only responsible for receiving the request and selecting an application server for forwarding; it does not do the actual request processing. The application server is responsible for the actual request processing and returns the processing results to the client side via the front-end server. Because the application servers do not accept user requests directly, their IP addresses are only visible to the front-end serve.

3.2. Concurrent Optimization Scheme of Front-End Server. The front-end server is an important node in the entire server cluster. It receives all user requests and forwards requests to the application server. Meanwhile, it is also responsible for the load balancing. In this paper, dual-machine hot standby technology is adopted to provide high availability for the front-end server. Two front-end servers are deployed, and one for the host and the other for the standby. When the host fails, the standby prepares to take over the task of the host quickly. When the host is repaired, the host can be serviced again. In the cluster scheme design, Nginx is used as the front-end server software. We design the connection scheme and load balancing algorithm for the front-end server.

3.2.1. Connection Scheme. The front-end server maintains two types of connections, one to clients and one to application servers in the cluster. Both connections use the HTTP protocol.

In the high concurrency environment, the communication between the front-end server and the client and the application server is relatively frequent. We should send as many HTTP requests as possible within a single TCP connection, that is, reusing the TCP connection to reduce the overhead of establishing and closing the connection. So, we adopt the HTTP long connection scheme. Nginx supports maintaining long connections with clients. Similarly, the connection between the front-end server and the application server is also the long connection scheme.

In addition, we apply the idea of connection pooling to manage TCP connections. The main steps are as follows: (1) when the front-end server starts up, it opens up a memory space for future establishing and opening TCP connections, which we call connection pooling. (2) After the front-end server decides which application server to forward the request to, it looks in the connection pool for a TCP connection that has the same target IP and port as the forward. If it is found, the TCP connection is taken out of the connection pool and it is used to forward the request. If not, a new TCP connection is created to forward the request. (3) When the front-end server receives the request processing results from the application server, the connection pool is checked, and the TCP connection which is used is saved for this request into connection pooling.

In the concurrent environment, the front-end server forwards requests to the application server very frequently, and the connection pool can be used to reuse TCP connections as much as possible to send requests to the same application server, which can reduce the overhead of frequently establishing and closing connections. At the same time, setting the maximum capacity of connection pool reasonably can avoid excessive memory consumption of the front-end server. To some extent, the scheme can avoid the slow start of TCP and increase the utilization of network bandwidth. Slow start is a method for TCP to estimate the available bandwidth of the network. After the establishment of TCP, the congestion window is set as 1; that is, the sender can only send one message at the beginning, and after receiving the acknowledgment from the receiver, the congestion window is gradually increased. Sometimes, too fast growth of the window will lead to a large number of packet loss. A multiplexed connection eliminates the need for the front-end server to start with a packet segment and reduces the chance of packet loss.

3.2.2. Load Balancing Algorithms. When forwarding a request, the front-end server uses a load balancing algorithm to select a back-end application server. In the cluster, the application servers are divided into two groups, one is used to handle simulation requests and the other to handle the query requests. The server selection is determined by load balancing algorithm [23, 24].

This paper proposes a dynamic load balancing algorithm combining polling. This algorithm will monitor the performance changes of each application server, then adjust the weight of each server according to the performance indicators, and finally select the forwarding server according to the weight.

The usage statuses of CPU and memory are used to measure performance of application server. Static

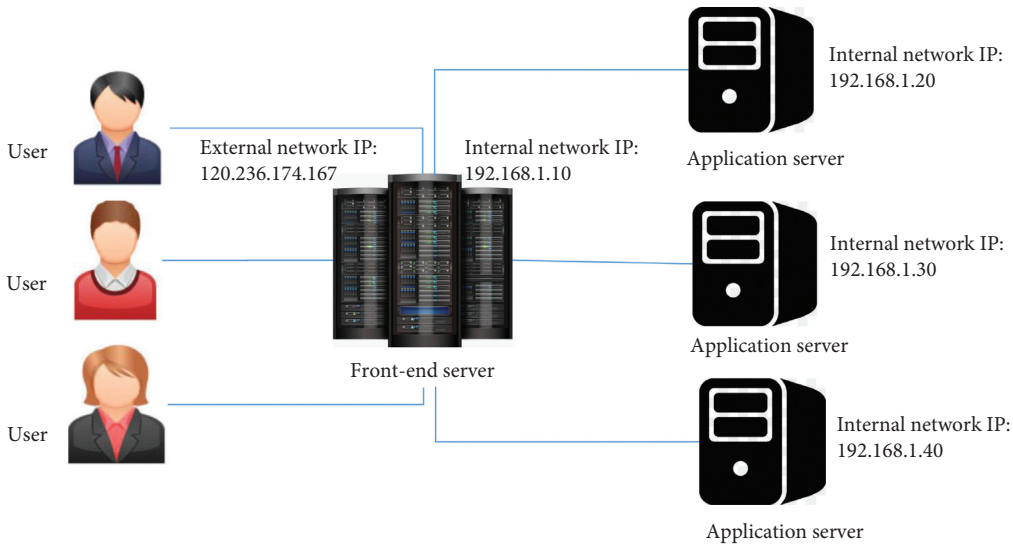


Figure 3: Cluster architecture.

performance of each application server is calculated based on its CPU configuration:

$$\text{performance}[i] = \text{frequency}[i] * \text{cores}[i], \tag{3}$$

where, performance $[i]$, frequency $[i]$, and cores $[i]$ represent the static performance values of the i_{th} application server, CPU dominant frequency, and CPU kernel number, respectively. We then set static weights for each application server based on static performance sw:

$$sw[i] = \frac{\text{performance}[i]}{\text{frequency}_{\min}} * 10. \tag{4}$$

After the front-end server is started, two weights are initialized for each application server: the current weight cw and the processing efficiency weight ew, where cw is initialized to 0 and initial value of ew is equal to the static weight.

Calculating the total remaining resources of the application server,

$$\text{totalResource} = \text{idleCPU} + \left(\frac{\text{freeMEM}}{200}\right), \tag{5}$$

where idleCPU is CPU vacancy rate and freeMEM is memory residual size. Update ew of the application server:

$$ew = \frac{(totalResource * \text{sw})}{10}. \tag{6}$$

After the front-end server receives the user request, the following steps are used to select an application server for forwarding:

(1) Update cw values for each application server, $cw = cw + ew$, and totalWeight is defined as the sum of the ew values for all the application servers

(2) Select the largest server on the cw

(3) Update the cw value of the selected server, $cw = cw - \text{totalWeight}$ and start forwarding request

The details are shown in Figure 4. In general, this algorithm combines the idea of polling and dynamic weight adjustment. cw is a key parameter to decide which server to choose. The larger the cw is, the more likely the server will be selected. ew reflects the server's ability to process requests at the current moment. In Step (1), the superposition of ew value makes the cw of the server with strong processing capacity grow rapidly and has a greater chance to be selected. In Step (3), the update operation makes the current request to choose the server of the cw decrease sharply, when forwarding a request under the chosen less likely, and this avoids multiple continuous forward requests to the same server. At the same time, if the server processing ability is very strong, the ew value is very high, which can bear more request; then, its cw can quickly recover in Step (1).

3.3. Concurrent Optimization Scheme of Application Server. In our server cluster, the application server is responsible for executing the real user request business. For a single request, the work flow of application server can be summarized as follows:

Step 1: monitoring the specified port and receiving the connection initiated by the front-end server

Step 2: reading the user request data forwarded by the front-end server from the connection and parsing the HTTP message

Step 3: finding the corresponding processing method based on the URL address and starting executing the actual business process

Step 4: when the business process completes, the processing results are sent back to the front-end server along the connection

In view of the threads blocking caused by sequential execution of the above steps, we divides application server threads into two categories. One is responsible for the first step operation, namely, receiving and establishing the

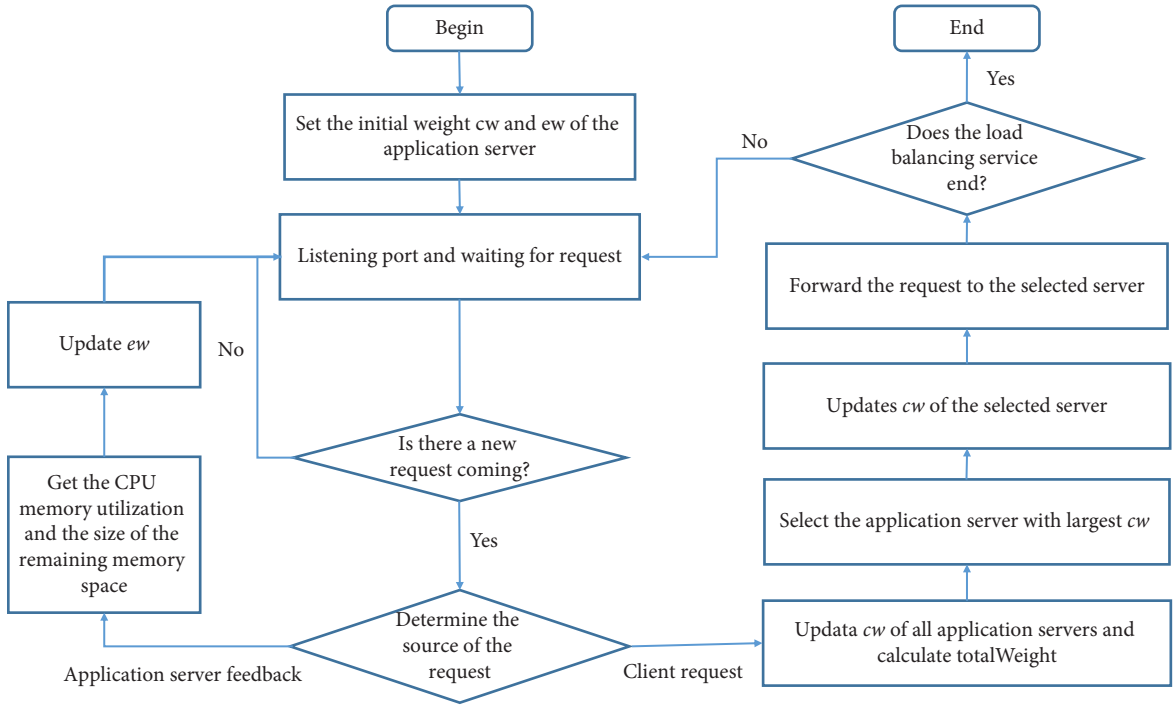


Figure 4: Flowchart of the dynamic load balancing algorithm.

connections, which is called the receiver thread. The other type of thread is responsible for Steps 2 to 4, that is, reading and writing request data, parsing messages, matching URL, executing business processes, etc., and we call it worker thread. In this way, after the receiver thread establishes the connection, it will create a socket for the connection and give it to the worker thread for subsequent processing. It will continue to listen on the port without blocking and can continue to receive new connections. The application server uses only one port to receive connection requests from the front-end server, so setting up a receiver thread is enough. We set the number of worker threads to be equal to the number of CPU cores in order to maximize the parallelism of multicore servers. Both types of threads use the epoll mechanism at the bottom to wait for IO

Figure 5 shows the thread model of application server designed in this paper. As shown in Figure 5, a business thread pool consists of several business threads and a task queue. minThreads and maxThreads are minimum number of threads and maximum number of threads of business thread pool. When the application server is initially started, the number of threads in the thread pool is 0 and the task queue is empty. The worker thread encapsulates the operation that actually handles the request as a task. There are three situations:

(1) If the current number of threads is less than maxThreads and all threads are running tasks, a new thread is created to perform the new task

(2) If the current number of threads is less than maxThreads, but there are idle threads that are not executing, then the new task is placed at the end of the task queue

(3) If the current number of threads is greater than or equal to maxThreads, the new task is placed at the end of the task queue

After each business thread is created, it is repeatedly to fetch the task from the task queue header and execute it in a loop, after that the result is asynchronously returned to the worker thread. If a business thread does not execute any tasks within the set timeout period, then the task queue is empty for a long time and the application server is idle. We can close the thread to save resources. However, if the number of business threads has been reduced to minThreads, we should not continue to close threads, because we need to maintain a certain number of threads to cope with the concurrent requests.

4. Case Study

To evaluate the effectiveness of the proposed scheme for server, we deploy a human sport health simulation model on the server that implements the concurrent optimization scheme.

4.1. Human Sport Health Simulation. With the improvement of people's health awareness, sports have become an indispensable part of life [25, 26]. However, if the exercise plan is not properly arranged, it is easy to cause physical discomfort. The human sport simulation model based on clothing heat and moisture transfer model can predict the physiological characteristics of human body during the exercise and further help people judge the change of body state and adjust the exercise plan reasonably [1]. At present, there is some software for sport health simulation [27], but

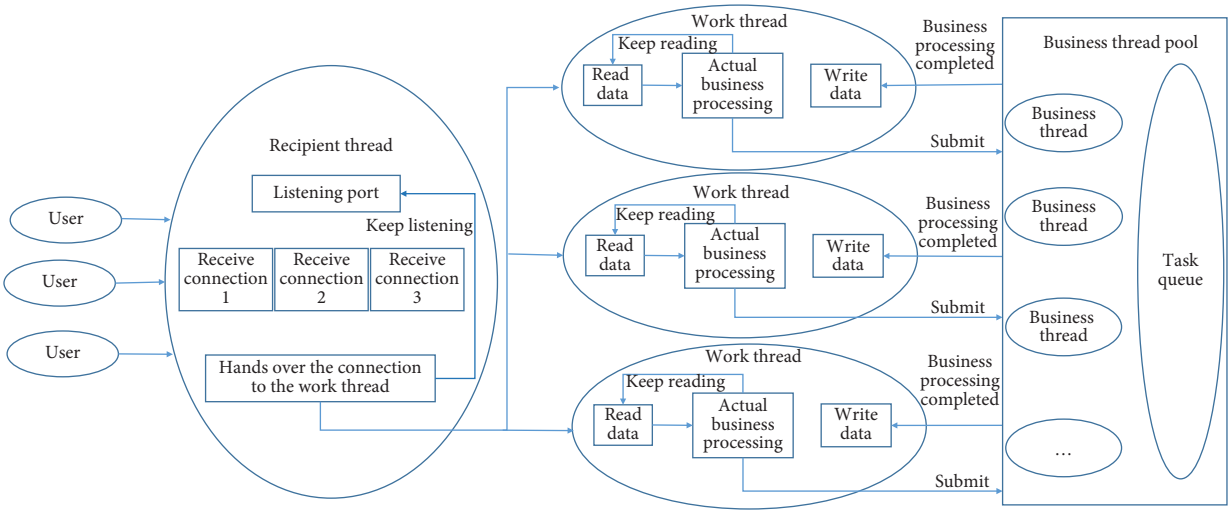


Figure 5: Thread model of application server.

the computation overhead of this software is large, as well as the parameter setting is complex, and the operation is cumbersome. Therefore, most of these software are deployed on the PC side, and they are mostly for professional researchers. It is difficult to popularize them to ordinary users. Therefore, it is of great significance to build an efficient and optimized sport health simulation platform and make it widely used in the field of the human sport.

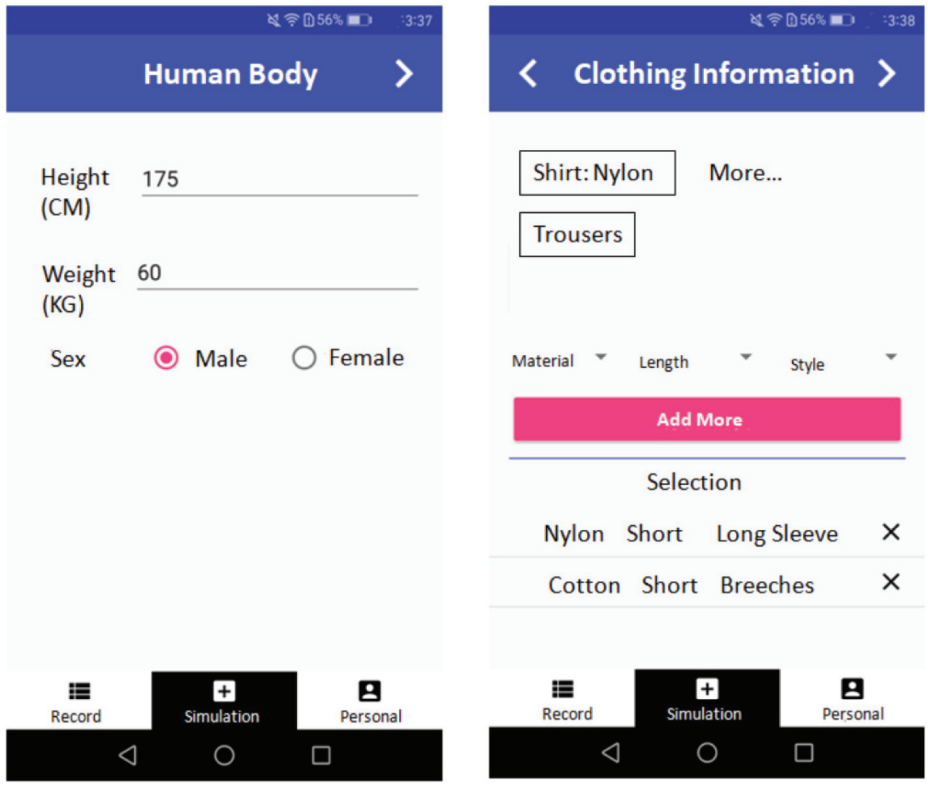
In this paper, we employ the 25-node human physical model to build the clothing heat and moisture transfer model. The human body in the 25-node model is divided into head, torso, arms, hands, legs, and feet (six parts), and each part from inside to outside is divided into the kernel, muscle, fat, and skin (four layers), plus central blood pool; therefore, it is called "25-node" [5]. The proposed human sport health simulation platform is divided into four modules: user information management, clothing management, simulation calculation, and simulation record management. User information management is responsible for executing the login and registration logic of the platform, saving the user's personal ID and password, and managing the user's height, weight and gender information, as shown in Figure 6(a). Clothing management is responsible for adding, deleting, checking, and modifying the user's preferred clothing, as shown in Figure 6(b). According to the relevant parameters provided by the user, the human sport simulation model is used to simulate the states of human during the exercise, and the change information of skin temperature and sweating amount during the user's exercise is returned in simulation calculation module. Simulation record management is responsible for saving the result data after each simulation process, and it can support users to query the previous sport health simulation results.

The human sport health simulation platform is deployed as C\S architecture. A friendly user interface is provided on the client app to facilitate the user to input personal information, clothing information, and exercise plan; the server employs the concurrent optimization schemes proposed in this paper, and we use the mature clothing heat and moisture transfer model and human thermal physiological regulation model to calculate the changes of human physiological characteristics [1]. When users start the simulation, they will set the simulation parameters on the mobile phone and upload them to the server that implements the concurrent optimization scheme, and the server will perform human sport simulation calculation according to those parameters. In general, the simulation process of this paper is an iterative solution process. We will iterate according to a certain step size within the duration set by the user's sport plan. Each iteration will involve the calculation of clothing heat and moisture transfer model and human thermal physiological regulation model.

4.2. Results Analysis

4.2.1. Application Server Performance. In this case, we use Netty framework (https://netty.io/index.html) to implement a lightweight servlet container (hereinafter referred to as the "article container") on the application server to replace the Tomcat container that spring boot uses by default. The concurrent optimization scheme designed in Section 3 is implemented in this container. Now, we test and compare the efficiency of this container and Tomcat container in processing requests in concurrent environment. The application server used in the test is configured with 3.4 GHz quad core CPU and 8 g memory. We deploy the server-side program of sport health simulation platform on the server.

We use JMeter (https://jmeter.apache.org/) to simulate the user to send simulation request. The parameters [23] of the request are shown in Table 1. We send concurrent requests to the server-side programs with this container and Tomcat container, respectively. Tables 2 and 3 show the throughput and average response time of the two containers when the number of concurrent requests per second is 100, 300, 500, 800, and 1000, respectively. Figures 7 and 8 show



(a) (b)

Figure 6: Client interface for inputting simulation parameters. (a) Human setting interface. (b) Clothing setting interface.

Table 1: Simulation settings and parameters.

Subject settings			Clothing settings		
Height (cm)	Weight (kg)	Sex	Material	Thickness	Style
170	60	Male	Cotton Nylon	Middle Middle	Short Trousers
Environment conditions			Exercise settings		
Temperature (°C)	Humidity	Type	Speed (m/min)	Duration (min)	
30	30%	Running	150	1	

Table 2: Throughput comparison between article container and Tomcat.

	Concurrent number				
Container	100	300	500	800	1000
Article container	42.9	40.4	40.8	39.5	40.3
Tomcat container	38.8	37.1	35.8	34.7	38.3

Table 3: Average response time comparison between article container and Tomcat.

	Concurrent number				
Container	100 (ms)	300 (ms)	500	800 (ms)	1000
Article container	914	3519	5836 ms	9576	11689 ms
Tomcat container	1806	5345	10066 ms	14897	15710 ms

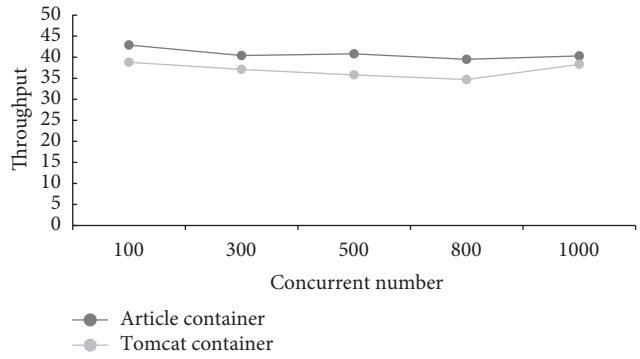


Figure 7: Throughput comparison of two containers.

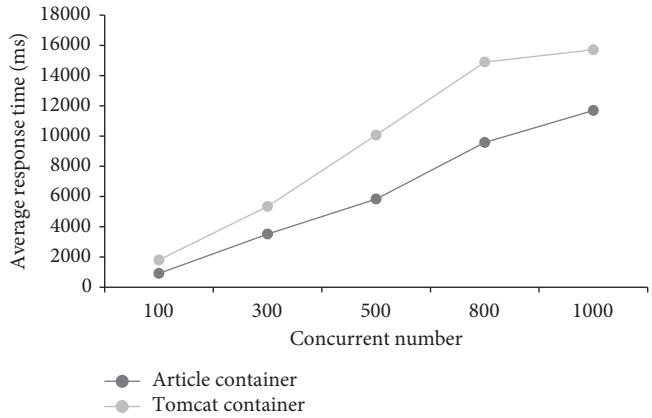


Figure 8: Average response time comparison of two containers.

the trend of throughput and average response time with the number of concurrent requests. Compared with the data in the figure, we can find that in the process of increasing the number of concurrent requests from 100 to 1000, the throughput of our container is higher and the average response time is shorter. Considering that 1000 concurrent simulation requests are already quite high concurrency for the server used in this experiment, we can think that the concurrency optimization scheme designed and implemented on the application server in this paper really improves the efficiency of the application server of the sport health simulation platform to process the requests concurrently.

4.2.2. Cluster Performance. Based on Nginx, we implement a dynamic load balancing algorithm and TCP connection pool, which combines polling and real-time performance on the front-end server. This section builds a cluster composed of a front-end server FS, a database server DS, and three application servers AS1, AS2, and AS3 for testing. The hardware configuration of each server is shown in Table 4. According to the calculation method in Section 3.2.2, the static weights for AS1, AS2, and AS3 are 11, 10, and 23, respectively.

Table 4: Hardware configuration of test machines.

	Configuration	
Machine	CPU	Memory (G)
FS	CPU frequency 3.20 GHz, dual core	2
AS1	CPU frequency 3.20 GHz, dual core	2
AS2	CPU frequency 2.93 GHz, dual core	2
AS3	CPU frequency 3.40 GHz, quad core	4
DS	CPU frequency 3.20 GHz, dual core	4

JMeter is also used to test in the experiment. The simulation user sends simulation request to the front-end server. The simulation parameters used are the same as those in Table 1. We use three schemes on the front-end server for comparative testing, which are the concurrent optimization scheme we proposed, the Nginx using polling (shorten for: Nginx Round Robin) and the Nginx using the least connection algorithm (shorten for: Nginx LBLC). The application server is used to deploy the human sport simulation model. Tables 5 and 6 show the cluster throughput and average response time of the three front-end server solutions when the number of concurrent requests per second is 500, 1000, 2000, 3000, and 4000, respectively. Figures 9 and 10 show the trend of the throughput and average response time

Table 5: Throughput comparison of three front-end server schemes.

	Concurrent number				
Scheme	500	1000	2000	3000	4000
Our algorithm	70.1	72.2	72.6	59.1	61.2
Nginx Round Robin	49.6	52.3	52.7	44.2	38.5
Nginx LBLC	66.8	64.6	65.3	56.8	54.9

Table 6: Average response time comparison of three front-end server solutions.

	Concurrent number				
Scheme	500 (ms)	1000 (ms)	2000 (ms)	3000 (ms)	4000 (ms)
Our algorithm	3093	6680	8455	11562	15514
Nginx Round Robin	3902	8508	14409	16707	19477
Nginx LBLC	3406	6695	9742	13528	17962

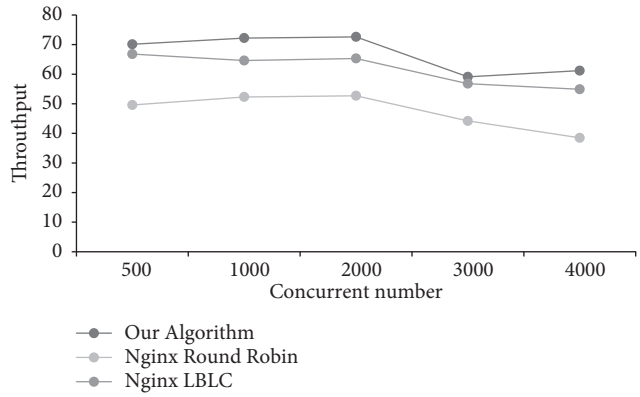


Figure 9: Throughput comparison of three schemes.

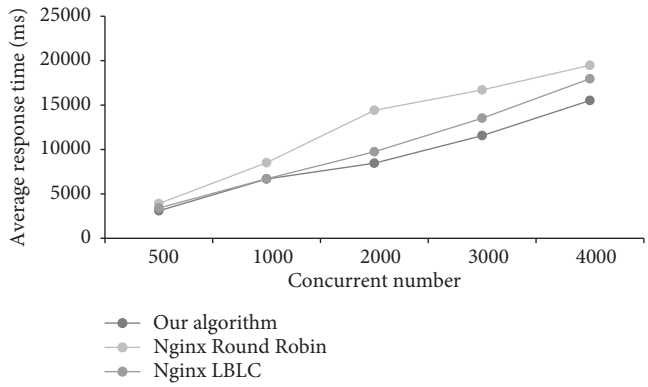


Figure 10: Average response time comparison of three schemes.

of the cluster with the number of concurrent requests. Compared with the data in the chart, it can be found that the cluster using the front-end server concurrency optimization scheme in this paper has higher throughput and shorter average response time. In addition, when the number of concurrency increases to 4500, no matter which algorithm is used, the cluster will start to have request failure. This is limited by the hardware configuration of the machine used in the experiment. Therefore, 4000 concurrent requests are very high for the server cluster used in this experiment. The experimental results show that the design and implementation of the front-end server concurrency optimization scheme, including the dynamic load balancing algorithm and TCP connection pool combining polling and real-time

performance, can effectively improve the efficiency of the server cluster of the sport health prediction platform to process concurrent simulation prediction requests.

5. Conclusions

In this paper, a concurrent optimization scheme for human body simulation is reported. It deploys a cluster of servers to handle concurrent user requests. This cluster uses the front-end server to receive and forward user requests and realizes the simulation model on the application server. The human exercise health simulation platform is designed and implemented on the server that optimized by the proposed concurrent scheme. The experiment results show that the concurrent optimization scheme designed and implemented in this paper can make better use of server resources and improve the processing efficiency of simulation applications in the face of concurrent simulation requests.

Data Availability

The data are being sorted out and will be released later.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The work was supported by the National Natural Science Foundation of China (nos. 61902105 and 61672547), Soft Science Research Project of Hebei Provincial Science and Technology Plan (no. 20550302D), and Key Research Project of Social Science Development of Hebei Province in 2019 (no. 2019021201005).

References

[1] N. Jia, X. H. Chen, L. Yu et al., "An exercise health simulation method based on integrated human thermophysiological model," *Computational and Mathematical Methods in Medicine*, vol. 2017, Article ID 9073706, 15 pages, 2017.

[2] D. W. Hensley, A. E. Mark, J. R. Abella et al., "50 years of computer simulation of the human thermoregulatory system," *Journal of Biomechanical Engineering*, vol. 135, no. 2, pp. 6–21, 2013.

[3] J. She, H. Nakamura, K. Makino, Y. Ohyama, and H. Hashimoto, "Selection of suitable maximum-heart-rate formulas for use with Karvonen formula to calculate exercise intensity," *International Journal of Automation and Computing*, vol. 12, no. 1, pp. 62–69, 2015.

[4] J. Jiao, *Effects of Clothing on Running Physiology and Performance in a Hot Condition*, The Hong Kong Polytechnic University, Hung Hom, Hong Kong, 2014.

[5] A. Mao, Y. Li, X. Luo, R. Wang, and S. Wang, "A CAD system for multi-style thermal functional design of clothing," *Computer-Aided Design*, vol. 40, no. 9, pp. 916–930, 2008.

[6] F. Z. Li, Y. Wang, and Y. Li, "A. transient 3-D thermal model for clothed human body considering more real geometry," *Journal of Computers*, vol. 8, no. 3, pp. 676–684, 2013.

[7] L. Peng, B. Su, A. Yu, and X. Jiang, "Review of clothing for thermal management with advanced materials," *Cellulose*, vol. 26, no. 10, 2019.

[8] M. V. Rodicheva, A. V. Abramov, E. M. Gneusheva, M. V. Rodicheva, A. V. Abramov, and E. M. Gneusheva, "Reducing the natural risk of the people working in the open area by clothing based on textile operating systems," *IOP Conference Series: Materials Science and Engineering*, vol. 962, no. 4, Article ID 042028, 2020.

[9] W. L. Kenney, J. Wilmore, and D. Costill, *Physiology of Sport and Exercise*, Human Kinetics, Champaign, IL, USA, 2015.

[10] T. Hamatani, A. Uchiyama, and T. Higashino, "Estimating core body temperature based on human thermal model using wearable sensors," in *Proceedings of the 30th Annual Symposium on Applied Computing, Salamanca, Spain: ACM*, pp. 521–526, New York, NY, USA, April 2015.

[11] M. Guan, S. Annaheim, M. Camenzind et al., "Moisture transfer of the clothing–human body system during continuous sweating under radiant heat," *Textile Research Journal*, vol. 89, 2019.

[12] N. Jia, Y. Huang, J. Li, H. An, X. Jia, and R. Wang, "Parallel simulation model for heat and moisture transfer of clothed human body," *The Journal of Supercomputing*, vol. 75, no. 8, pp. 4731–4749, 2019.

[13] Y. H. Wang, *High Performance Web Server Performance Optimization Based on Nginx and Load Balancing Improvement and Implementation*, University of Electronic Science and Technology of China, Chengdu, China, 2015.

[14] R. Will, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 173, p. 2, 2008.

[15] J. L. Abitbol, E. Fleury, and M. Karsai, "Optimal proxy selection for socioeconomic status inference on Twitter," *Complexity*, vol. 2019, no. 5915, 15 pages, 2019.

[16] D. R. Karger and M. Ruhl, "Simple efficient load balancing algorithms for peer-to-peer systems," *Theory of Computing Systems*, vol. 39, no. 6, 2004.

[17] M. Elgili, "Load balancing algorithms round-robin (RR), least-connection and least loaded efficiency," *International Journal of Computer and Information Technology*, vol. 4, no. 2, pp. 255–257, 2015.

[18] C. H. Qu, R. N. Calheiros, and R. Buyya, "Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing: mitigating overload on multi-cloud web applications through GLB," *Concurrency and Computation Practice and Experience*, vol. 29, no. 1, Article ID e4126, 2017.

[19] E. Daniel, Y. Cheng, C. Contavalli et al., K. J. Argyraki and R. Isaacs, Maglev: a fast and reliable software network load balancer," in *Proceedings of 13th USENIX Symposium on Networked Systems Design and Implementation*, pp. 523–535, USENIX Association, Santa Clara, CA, USA, March 2016.

[20] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 9, pp. 979–993, 1993.

[21] X. Chi, B. Liu, and Q. Niu, "Web load balance and cache optimization design based Nginx under high-concurrency environment," *Journal of Physics A Mathematical & Theoretical*, vol. 45, no. 48, Article ID 485305, 2012.

[22] L. Gammo, T. Brecht, A. Shukla et al., "Comparing and evaluating epoll, select, and poll event mechanisms," in *Proceedings of the 6th Annual Ottawa Linux Symposium*, Ottawa, Canada, July 2004.

[23] G. Zhang, H. Liu, P. Li et al., "Load prediction based on hybrid model of VMD-mRMR-BPNN-LSSVM," *Complexity*, vol. 2020, no. 4, 20 pages, 2020.
[24] J.-P. Yang, "Elastic load balancing using self-adaptive replication management," *IEEE Access*, vol. 5, pp. 7495–7504, 2017.
[25] P. J. Vanbeveren and D. Avers, "Exercise and physical activity for older adults," *Geriatric Physical Therapy*, vol. 41, pp. 64–85, 2012.
[26] M. E. Nevill, "Exercise physiology," *Bulletin British Association of Sport & Medicine*, vol. 25, no. 4, pp. 247–259, 2005.
[27] H. Gjoreski, B. Kaluža, M. Gams, R. Milić, and M. Luštrek, "Context-based ensemble method for human energy expenditure estimation," *Applied Soft Computing*, vol. 37, pp. 960–970, 2015.