IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 6, 2022, accepted January 23, 2022, date of publication January 26, 2022, date of current version February 8, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3146422

Evaluation Test and Improvement of Load Balancing Algorithms of Nginx

CHEN MA AND YUHONG CHI
School of Information Engineering, Xijing University, Xi'an 710123, China
Corresponding author: Chen Ma (997528107@qq.com)

ABSTRACT Today most people can't live without electronic devices. And more and more new devices are coming to the market every day. Apps running on these devices often connect to one or several web-based server side applications, which in turn put a lot of load and management pressure on the servers and clusters serving these web applications. Technologies such as Nginx and Keepalived were invented to address the load issues faced by these high concurrency applications. This paper tested a server cluster environment based on Nginx and Keepalived, evaluated the performance of Nginx based algorithms such as WRR, IP_HASH and LEAST_CONN, and designed an optimized version of IP_HASH (named as NEW_HASH). Compared to the original IP_HASH, the NEW_HASH reduces the probability of hash collisions and improves the performance of searching back-end nodes. The test showed NEW_HASH outperforms the original method of IP_HASH, with reduced response time, lower failure rates and increased throughput. Overall, the server cluster performed better under the high load pressure using NEW_HASH.

INDEX TERMS High concurrency, IP_HASH, load balancing, Nginx.

I. INTRODUCTION

In recent years, lots of internet based software flooded into people's daily lives. These kinds of software satisfy the daily needs of many people. They also received a lot of visits and generated a lot of data, which puts a lot more pressure on managing and safeguarding the servers serving these applications. Page views of short videos, music programs, and all kinds of photos information have been increasing. The single server system can no longer support the demands of these applications due to increased demands for availability and reliability.

The server clusters can solve problems caused by the too higher load. Nginx is a web server that can also be used as reverse proxy and load balancer. Used as a reverse proxy, Nginx can make a proxied object become a new server, ease the workload and make caches of pages visited often. It also can forward requests to the target server behind a proxy server, then return the data from the server back to the sender through websites. Both the proxy server and target server can be on the same physical machine.

Nginx can also send requests to connect the most suitable server in the current cluster by modifying the profile. Native algorithms of load balancing in Nginx include RR (The Round_Robin), WRR (Weighted_Round_Robin), IP_HASH, and LEAST_CONN, etc. For different application environments, every algorithm has its best operating environmental conditions. In reference [1], this paper mainly introduces a new load balancing model based on the genetic method. The design of the model and the construction of parameters are introduced and explained in detail. Various environmental impact factors are considered in theory, and reliable results are obtained through several experimental tests. In reference [3], this paper created a new traffic path planning algorithm based on data prediction (TPPDP), which can find the path with the shortest travel time. This algorithm built a specific distributed computing framework to reduce the algorithm's runtime and optimize the load balancing effect. In reference [8], this paper created a new balancing algorithm for cloud computing applications, optimizing resources and improving the load balancing in task parameters, the priority of virtual machines, and resource allocation. It analyzes various performance parameters in the same cloud distributed virtual environments, such as average makespan and average execution time, and analyzes the effect of performance parameters in different virtual environments. The algorithm results are analyzed from two aspects, which makes the conclusion more convincing. In reference [12], this paper created an improved switch migration decision algorithm (ISMDA) that solved the network challenge when the load is elephant flow. The performance comparison metrics include the controller throughout, response time, number of

The associate editor coordinating the review of this manuscript and approving it for publication was Hongwei Du.

migration space, and packet loss, and this algorithm constructed a relatively complete measurement model.

For the IP_HASH method in Nginx, its native hash function is too simple, leading to a high possibility of hash collision. In addition, load balancing algorithms mentioned in the above references do not provide any practical and feasible methods to solve the problem of hash collision that may occur in the back-end cluster. Therefore, this paper aims to reduce the probability of hash collision in the back-end cluster by optimizing and modifying the hash function in the IP_HASH method. This paper mainly compares and evaluates different situations under different algorithms in the cluster based on the load balancing of Nginx and the high availability of Keepalived. We evaluated original algorithms in Nginx such as WRR, IP_HASH, and lEAST_CONN, and designed a new optimized method based on the IP_HASH, which can increase the efficiency and the speed of searching. We called it NEW_HASH.

II. RELATED TECHNOLOGIES AND ALGORITHMS

This paper's algorithm test and evaluation are built on a relatively stable Web cluster environment. The following technologies are necessary for the test environment construction in this paper.

A. FORWARD PROXY

Before realizing the load balancing, we first need to understand the concepts of forward and reverse proxies. Its principle is shown in the diagram below:

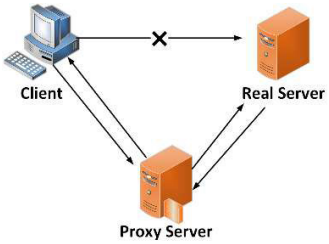


FIGURE 1. The principle of forward proxy.

It can be seen from Fig. 1 that the forward proxy is similar to a stepping stone, and it can access sources outside as a proxy server. For example, if the current client cannot access one website, another service machine can do. We can just access the desired website and accept the data we cannot get before via this service machine. In summary, the forward proxy is in charge of the client-side. The forward proxy has the following features:

1.The current clients can accept the data they could not get before by accessing the proxy server.

2.Forward proxy can ensure the safety of the internet protocol address of the real server.

3.But once the proxy server breaks down or delays, the clients will lose the access.

B. REVERSE PROXY

After knowing the concepts of the forward proxy, the principle of reverse proxy can be easy to understand. Its principle is as follows:

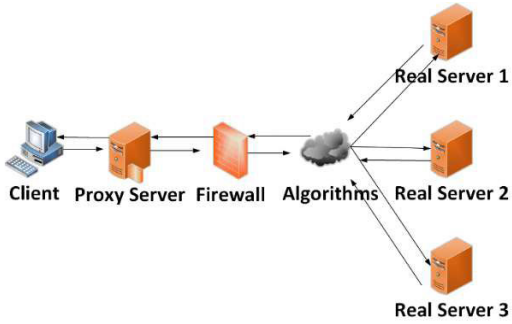


FIGURE 2. The principle of reverse proxy.

Fig. 2 shows that the internet protocol address of each server being proxied in one cluster is invisible to clients. However, clients only need to access the proxy server, and then it will send the requested data back to the client. Choosing which back-end server in the cluster depends on the load balancing algorithm covered in this paper. In other words, the reverse proxy is in charge of the server-side. It has the following features:

1.Actual internet protocol addresses of servers being proxied are invisible from the outside. And only the internet protocol addresses of proxy servers can be seen. So then, the reverse proxy can ensure the safety of actual servers.

2.Reverse proxy can ensure the safety of internet protocol addresses and the concurrency of the cluster.

3.Reverse proxy also solves the problem of information overload in the single point server.

C. LOAD BALANCING OF NGINX

The load balancing of Nginx is based on the reverse proxy.

The load balancing is to calculate which server is most suitable for the cluster environment of the back end when users access the back end. Its principle is as follows:

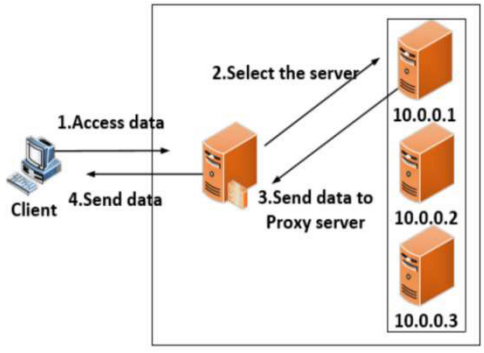


FIGURE 3. The principle of load balancing.

As shown from Fig. 3, the client will first request data to the proxy server, and then the proxy server will select the appropriate back-end server by using the load balancing algorithm. Finally, the back-end server will pass the data that

the user requests back to the client. In the whole process, the reverse proxy ensures that the real IP addresses of servers are not visible to the client.

The load balancing of Nginx can also solve the problem that the server may break down because of too many accesses. To give clients a better access experience, Nginx can set up a server cluster. And when users access this cluster, the load balancing can use different algorithms to adapt to different network environments. Then it will help you connect to the server which is best suited to the current network environment or which will result in the minimum load in this cluster.

For example, we can use the upstream module in 'nginx.conf' to set up three servers: 192.168.88.129:8080, 192.168.88.129:8081, and 192.168.88.129:8082. Then we assign the weight proportion to each of them depending on the current cluster environment, like 2:1:1. These operations are steps to configure the algorithm WRR. This way, we can ensure that the load pressure in the current cluster environment to be relatively stable in case the individual server crashes.

D. HIGH AVAILABILITY

After configuring the reverse proxy and load balancing in the current cluster, there is also a problem in the current environment——one single proxy server may not be safe. The actual internet protocol addresses in servers can be invisible from the outside, and the loads are balanced, for all the servers being proxied. Once the single proxy server breaks down, the clients can no longer access the real servers. Therefore, Keepalived realizes the high availability technology, which provides a health check based on VRRP (Virtual Router Redundancy Protocol) and recognizes fault transformation among many server machines.

The principle of high availability is as follows:

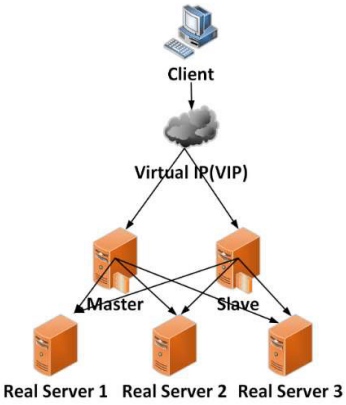


FIGURE 4. The principle of high availability.

Fig. 4 shows two different proxy servers in this situation, one is a master scheduler, and the other is a slave scheduler. They are bound to the same virtual internet protocol address named VIP. Then we can set their priority respectively (the priority of master scheduler is higher than the priority of slave scheduler generally), and there is also preemptive scheduling in the configuration of Keepalived. When Keepalived starts, VIP will be bound to the master scheduler, and users can accept data from the suitable server in the current cluster. But when it breaks down, VIP will be attached to the slave scheduler to ensure the continuity of data reception. When the master scheduler comes back, VIP can also be bound back to it. High availability is all about preventing the proxy server from going down and making the whole cluster inaccessible.

The above four mentioned technologies are also the technologies for constructing the experimental environment involved in this paper and the background basis for the effects of load balancing algorithms and software technical supports required for the actual projects.

E. NATIVE ALGORITHMS OF NGINX

In this article, in addition to the innovative algorithm, we also need to mention the following three algorithms for comparison, namely WRR, IP_HASH, and LEAST_CONN. The following table provides the notations required to describe the following three methods:

TABLE 1. Main notations used in WRR, IP_HASH, and LEAST_CONN.

Notation	Definition
CWeights[ ]	An array of current weights in the cluster
EWeights[ ]	An array of effective weights in the cluster
MaIndex	Index of the max weight
Nodes[ ]	An array of current nodes in the cluster
SeNode	A selected node in the current loop
count_nodes	Number of current nodes in the cluster
total_weight	Sum of initial weights in the cluster
IP_Addr[ ]	An array of IP addresses of the client
Hash()	Hash function
hash	Value of hash
w	*hash % total_weight*
peer	Pointer to array
weight	Weight of a single node
p	Index of a single node
best	A selected node in the cluster
conns	Active connections of a node
many	Flag of back-end nodes with same conditions

1) WRR

The full name of this algorithm is Weighted_Round_Robin. This algorithm can define the access frequency of each server by configuring the weight after the status of the proxy server is known, and it can reasonably allocate access ports according to its weight ratio. The algorithm of the WRR is as follows:

In ALGORITHM 1, *CWeights[ ]* starts with 0 values. The values of *EWeights[ ]* are customized in the configuration file based on the current cluster environment. Before the algorithm runs, the first step is to add the corresponding value of *EWeights[ ]* to *CWeights[ ]*(lines 1-2), then the value of *total_weight* is the sum of weights of *EWeights[ ]*(lines 3). For example, the value of *EWeights[ ]* is 2, 4, and 6, and

Algorithm 1 Weighted Round Robin

Input: *CWeights[ ], EWeights[ ], Nodes[ ], count_nodes, MaIndex, total_weight*
Output: *SeNode,*
1: Initialize: **for** i **in** *count_nodes:*
2: **Add** *EWeights[i]* **into** *CWeights[i]*
3: *total_weight* + = *EWeights[i]*
4: **while(true)**:
4: **for** j **in** *count_nodes*:
5: *Maindex* = *CWeights[0]*
6: **if** *Maindex* < *CWeights[j]*
7: Maindex = CWeights[j]
8: **end if**
9: *SeNode* = *Nodes[Maindex]*
10: *CWeights[Maindex]* -= *total_weight*
11: **for** k **in** *count_nodes:*
12: **Add** *EWeights[k]* **into** *CWeights[k]*
13: **Return***SeNode*

Algorithm 2 IP_HASH

Input: *IP_Addr[ ], EWeights[ ], count_nodes, hash, total_weight, w, peer, p, Nodes[ ]*
Output: *SeNode*
1: Initialize: **for** i **in** 3:
2: *hash* = *Hash(IP_Addr[i])*
3: **for** j **in** count_nodes:
4: *total_weight* + = *EWeights[i]*
5: *w* = *hash* % *total_weight*
6: *peer* = *EWeights[0]*
7: *p* = *0*
8:**while**(*w* >= *peer* -> weight):
9: *w* -= *peer* -> *weight*
10: *peer* = *peer* -> *next*
11: *p* + +
12: *SeNode* = *Nodes*[*p*]
13: **Return** *SeNode*

their index is 0, 1, and 2, respectively. Firstly, we can calculate that the values of *CWeights[ ]* are also 2, 4, and 6(lines 1-3), then this algorithm will select the server node of the maximal weight in this cluster(lines 4-8), and we can know that the total weight is 12(lines 3). Secondly, we choose the node with the highest value(*SeNode*) to be the node visited at this time(lines 9- 10, 13). Then the maximal weight (*CWeights[Maindex]*) will subtract the total weight(lines 10), and we can get a new list called effective weights: 2, 4, and -6. Finally, we can get the sum of each weight of *CWeights[ ]* and *EWeights[ ]*: 4, 8, and 0(lines 11-12), and this algorithm will continue the same loop with a new *CWeights[ ]* one by one(lines 4).

It can be seen that the WRR polls all the server nodes in the current load cluster and calculates all current weights in them, then select the server node of the maximal weight and calculate the sum of effective weights and current weights for the next loop. So we can know the applicability of this algorithm: after the type and performance of the load balancing server are known, the load balancing effect can be achieved by setting heavy weights to servers with high load capacity and small weights to servers with low load capacity. It is used when the performance of each back-end server is unbalanced or just setting different weights in the case of the master and slave server to achieve reasonable and effective use of host resources.

2) IP_HASH

The IP_HASH technology directs requests from an IP address to the same back-end web machine so that a client under that IP address and a back-end web machine can establish a solid session. The algorithm flow of the IP_HASH is as follows:

In ALGORITHM 2, during initialization, the algorithm firstly calculates a hash value of the first three digits of the *IP_Addr[ ]* by using the *Hash( )*(lines 1-2), then it calculates the value of *total_weight*, which is the sum of weights of *EWeights[ ]*(lines 3-4). Secondly, the *w* is the remainder of *hash % total_weight*(lines 5), and the peer is pointed to the beginning of the array *EWeights[ ]*(lines 6). Finally, the algorithm traverses back-end nodes, subtracts the weight of each node by *w* in turn, and calculates the index of the back-end node where *w* is less than its weight (lines 8-12). This time, the SeNode is the selected node, and then the algorithm will continue the next loop.

The above-mentioned shows that this is an algorithm that uniformly forwards packets from the same sender (or packets destined for the same destination) to the same server by managing the hash of the sender and destination IP addresses. When the client has a series of services to be processed and must communicate with a server repeatedly, the algorithm can take the flow (session) as the unit to ensure that the communication from the same client can always be processed in the same server, and maintain the continuity of user requests and requested session information.

So if you want to improve the performance of IP_HASH, the modification of the hash function is very important, which is the starting point of this paper to focus on transformation and improvement.

3) LEAST_CONN

First of all, we need to understand that the Round Robin algorithm is to evenly forward requests to each back end so that their load is roughly the same. And this assumes that each request takes about the same amount of back-end time, and if some requests take a long time, the back-end load is high. In this scenario, forwarding requests to the back end with fewer connections can achieve a better load balancing effect, which is the advantage of the LEAST_CONN. The algorithm flow of the LEAST_CONN is as follows:

In ALGORITHM 3, the algorithm firstly traverses the back-end cluster to compare the value of *conns/weight* and select the smallest (lines 3-4). If only one node can meet this condition in the current cluster, it will be the best selected in

Algorithm 3 LEAST_CONN

Input: *EWeights[ ], count_nodes, best, peer, Nodes[ ], many*
Output: *SeNode*

```
1: Initialize: best = null
2:             peer = Nodes[0]
3: for i in count_nodes:
4:       if (best == null) or ((peer -> conns * best ->
5: weight) < (best -> conns * peer -> weight))
6:             best = peer
7:             many = 0
8:             p = i
9:             SeNode=Nodes[p]
10:      else if((peer -> conns * best -> weight) ==
11: (best-> conns * peer -> weight))
12:            many = 1
13:      end if
14: if many:
15:      use the function of WRR
16: Return SeNode
```

the current loop (lines 3-9). On the other hand, if multiple nodes can meet the condition, the algorithm will choose one using the function of WRR (lines 14-15).

In addition, we can see from Algorithm 3 that the concept of LEAST_CONN is not a simple sum of real connections, and the effect of LEAST_CONN will be different with the user-defined weight. For example, if Server1 has 100 connections and Server2 has 80 connections, since the weights are 25 and 40 respectively, the results of *conns/weight*s are 4 and 2. So the cluster will select Server2 over Server1 because $2 < 4$, even though the number of connections on Server1 is bigger than that on Server1. So the LEAST_CONN is not what it literally means.

III. DESIGN AND IMPLEMENTATION OF THE NEW_HASH ALGORITHM

After introducing the three algorithms mentioned above, this section mainly introduces the new algorithm's theoretical logic and implementation steps based on IP_HASH. The optimized algorithm mentioned in this paper is NEW_HASH (improvements are all in the Nginx configuration file ngx_http_upstream_ip_hash_moudle.c).

There are two innovations in NEW_HASH based on the source code of IP_HASH:

A. THE INNOVATION OF THE HASH FUNCTION

The source code of the hash function of IP_HASH is as follows:

At first, this old hash function in IP_HASH is rough and straightforward to calculate hash numbers by using the first three digits of internet protocol addresses. And in many algorithms of hash, the hash situation of the remainder calculation is not good because the hash collision of this kind of algorithm is apparent. The new change on the hash function is as follows:

```
static ngx_int_t
ngx_http_upstream_get_ip_hash_peer
(ngx_peer_connection_t *pc, void *data){….
for(;;){
for(i=0;i<(ngx_uint_t)iphp->addrlen;i++){
/*The hash number of the source code*/
hash=(hash * 113+iphp->addr[i]) % 6271);}
….}….}
```

FIGURE 5. The hash function of IP_HASH.

```
static ngx_int_t
ngx_http_upstream_get_ip_hash_peer
(ngx_peer_connection_t *pc, void *data){....
for(;;){
for(i=0;i<(ngx_uint_t)iphp->addrlen;i++){
/*The new hash function*/
hash=(hash * 33+iphp->addr[i]) ^ (hash >> 16));}
....}....}
```

FIGURE 6. The new hash function.

The inspiration of this design is from the source code of HashMap in Java:

```
static final int hash(Object key){
        int h;
        /*The method of HashMap*/
        return (key == null)?0:(h=key.hashCode() ^
(h >>> 16));}
```

FIGURE 7. The hash function of HashMap in Java.

Fig. 6 and Fig. 7 show that the first step of the new hash function is to change 113 to 33, which means mainly to transform a prime number, which continues the hash function feature of Times 33.

Times 33 is a hash algorithm that Daniel J. Bernstein published in comp.lang.c many years ago, and it's one of the best hash algorithms out there because it's fast and evenly distributed. In other words, using the number 33 can make the hash better and faster. So why it works better than many other constants, prime or not? Daniel J Bernstrin explained that if one tests all multipliers between 1 and 256, one will detect that even numbers are not useful. The remaining 128 odd numbers (except for the number 1) work more or less all equally well. If one compares the Chi-SquareStatistic of the variants, the number 33 is not the best value. But the number 33 and a few other equally good numbers like 17, 31, 63, 127, and 129 have a great advantage nevertheless to the remaining numbers in the large set of possible multipliers: their multiply operation can be replaced by a faster operation based on just one shift plus either a single addition or subtraction operation. The purpose of Times 33 is to increase the hash degree and improve the hash efficiency.

The second step of the new hash function is moving the unsigned bit of the hash 16 bits to the right and doing the XOR operation. The principle of this is as follows:

1: The 16-bit shift to the right reduces hash collisions further. For example, the value of int is 4 bytes, and a 16-bit change to the right and XOR (Exclusive OR) can preserve the characteristics of the upper and lower 16 bits. This can be shown as follows:

```
                0010-0101 1100-0100 0010-0101 1011-0110
XOR(^)          0000-0000 0000-0000 1010-0101 1100-0100
-------------------------------------------------------
                0010-0101 1100-0100 1000-0000 0111-0010
```

FIGURE 8. XOR operation.

Fig. 8 shows that the XOR operation can mix the features of the highest 16-bit with the lowest 16-bit as much as possible, resulting in a hash function with less repetition. So why did we select the XOR operation? First, because the result of AND operation is close to 0, and the OR operation is close to 1, only the XOR operation can retain characteristics of various parts. And then, in this way, when the number of requests is too large, the different Internet protocol address can get their hash number as soon as possible. So they can access other servers in the current cluster to reduce the workload pressure of the current environment.

2: Compared with the hash function shown in Fig. 5, the probability of hash collision caused by XOR operation is obviously much smaller, as shown in the following figure:

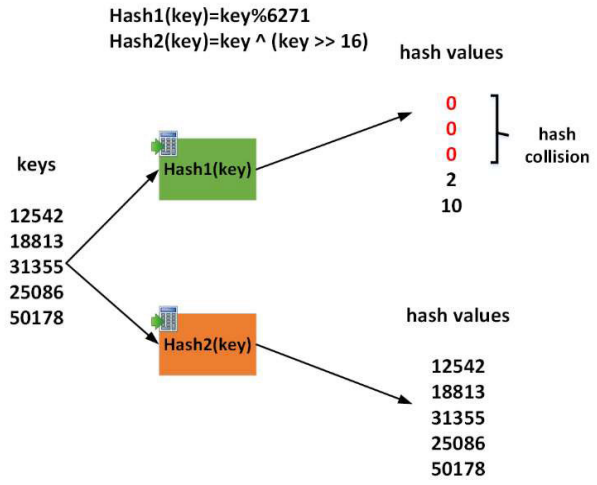


FIGURE 9. The comparison of two operations.

It can be seen from Fig. 9 that moving the unsigned bit of the hash 16 bits to the right and doing the XOR operation really reduce the probability of hash collisions. Compared with the old hash function of IP_HASH, the innovation in the body of the hash method not only refers to Times 33, which is more reasonable in function selection but also chooses a better operation method in function operation.

B. THE INNOVATION OF SEARCHING WAY

The source code of the searching function of IP_HASH is as follows:

```
static ngx_int_t
ngx_http_upstream_get_ip_hash_peer
(ngx_peer_connection_t *pc, void *data){....
{   w=hash%iphp->rrp.peers->total_weight;
    for(i=0;i<iphp->rrp.peers->number;i++){
        w-=iphp->rrp.peers->peer[i].weight; }
    if(w<0){break;}....p=i;}....}
```

FIGURE 10. The searching function body of IP_HASH.

Fig.10 shows how the IP_HASH finds the most suitable node in the current loop, and we can also find the corresponding algorithm flow in ALGORITHM 2(lines 7-13) in Section 2. From the body of the searching function, there are also some problems in specific application environments: if there are too many server nodes in the current cluster, and the number of w is too high, every weight of every node is small, it will take a lot of loops to find the correct index. This kind of searching way is also named sequential search, whose time complexity is O(n) (n is the length of a sequence sorted). It can meet most simple cluster test tasks, but once applied to the multi-node and complex load cluster environment, this searching algorithm will bring too much loading pressure to the load cluster environment when the number of clients is too high. To adapt to the pressure of a more realistic load cluster environment, we can optimize this body of the searching position. And this new part is shown in the algorithm flow chart:

It can be seen from Fig. 11 that there is a newly created array in global variables, and firstly we put every weight of every server node into it when iterating through the first three digits of internet protocol addresses. Then we sort the data in the array using quicksort and find the node that matches the criteria by using binary search, and finally, the array's space needs to be free. It is necessary to sort the array of weights before binary search because binary search is a searching algorithm that looks for a specific element in an ordered array. Therefore, in the case of disordered elements of the weight array, it is necessary to sort them to facilitate the effectiveness of binary search. The following table provides the notations needed to describe the quicksort and binary search:

Next, we need to introduce the concepts and principles of quicksort and binary search:

1) QUICKSORT

The basic idea of quicksort is that the records to be sorted are separated into two independent parts through a sort. If the keywords of one part of the records are smaller than those of the other, the two parts of the records can be sorted separately to achieve the order of the whole sequence. The algorithm flow of the quicksort for NEW_HASH is as follows:

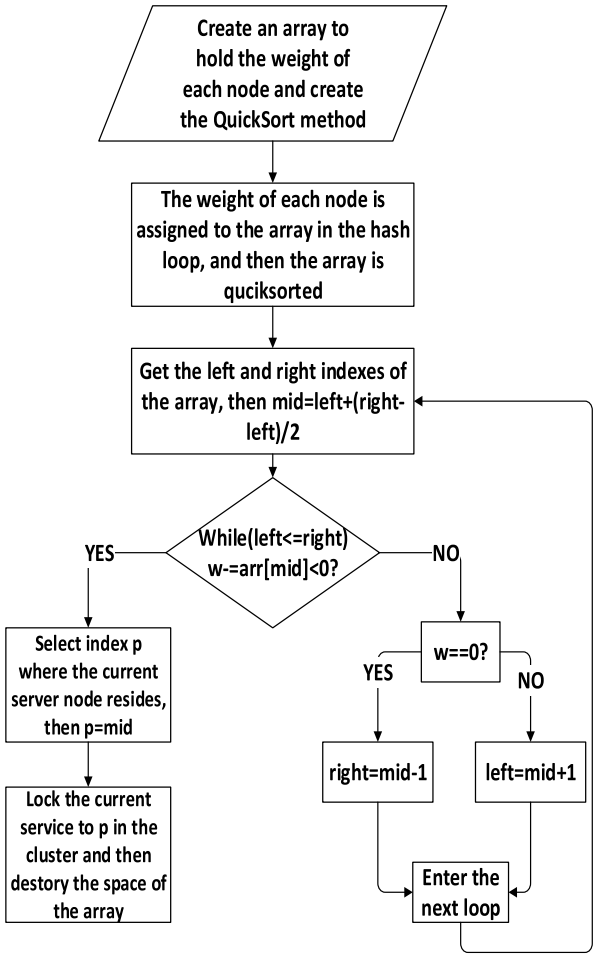


FIGURE 11. The new searching algorithm flow.

TABLE 2. Main notations used in Quicksort and Binary Search.

Notation	Definition
arr[ ]	An array containing weights of nodes
low	Pointer to an array
high	Pointer to an array
k	Baseline data for each sorting
QuickSort( )	Function of quicksort
arr_size	The length of data stored in *arr[ ]*
left	Pointer to an array
right	Pointer to an array
free( )	Destroy an array

In ALGORITHM 4, during initialization, the algorithm set two variables: *low* and *high*, when sorting starts: *low* = 0, *high* = size-1(lines 1-2). The first element of the default array is the baseline data(lines 3), and then the algorithm starts to sort in a loop:

1) Since the first element of the default array is the baseline data, we search from the back, find the first *arr[j]* less than *k,* and assign *arr[j]* to *arr[i++]*(lines 4-9).

Algorithm 4 Quicksort

Input: *arr[ ], low, high*

```
1: Initialize:  i = low = 0
2:              j = high = (arr_size - 1)
3:              k = arr[low]
4: while (i < j):
5:while (i < j and arr[j] >= k):
6        j–
7:      if (i < j):
8:          arr[i++] = arr[j]
9:       end if
10:   while (i < j and arr[j] < k):
11:           i++
12:        if (i < j):
13:          arr[j–] = arr[i]
14:       end if
15: arr[i] = k
16: QuickSort(arr, low, i - 1)
17: QuickSort(arr, i + 1, high)
```

2) Secondly, we search from the front, find the first *arr[i]* greater than *k,* and assign *arr[i]* to *arr[j–]*(lines 10-14).
3) Then we repeat steps 1-2 until low = high, which is the index of the baseline data in the next loop.
4) Finally, the next step is assigning the baseline data to the current index (lines 15).
5) The next step is to quickly sort the first and last parts separated from the array by the baseline data and follow steps 1-4, and we can get an array in ascending order.

After understanding the realization process of the quicksort, we can get the following conclusions: first of all, it is a sorting algorithm, and sorting algorithms are designed to turn disordered combinations of data into ordered combinations of data, whose most significant advantage is that it is very convenient for you to locate and use data. And this will save you a lot of unnecessary trouble when designing your codes. Secondly, if the weight of each node is sorted by quicksort before the binary search, it will also help binary search find the appropriate back-end node better. Third, when the quicksort is applied to a truly high-concurrency environment, it will actually help the current load cluster increase data access efficiency and reduce the burden of the query.

2) BINARY SEARCH

In an ordered list, a sequential search requires at most N comparisons from one end to the other (N is the maximum length of the list). The performance of sequential search drops as the value of N increases. In NEW_HASH, we combined quick sort and binary search to improve the search efficiency. The algorithm flow of NEW_HASH is as follows:

In ALGORITHM 5, during initialization, the algorithm set three variables: *left*, *right*, and *hash*. When searching starts: *left* = 0, *right* = *count_nodes*-1, and *hash* = 89 (lines 1-3). It then starts iterating through the back-end nodes, getting the

Algorithm 5 NEW_HASH

Input: *IP_Addr[ ]*(In TABLE 1), *arr[ ], left, right, count_nodes*(In TABLE 1), *w*(In TABLE 1), *hash*(In TABLE 1), *EWeights[ ]*(In TABLE 1), *total_weight*(In TABLE 1), *Nodes[ ]*(In TABLE 1), *SeNode*(In TABLE 1)
Output: *p*(In TABLE 1)

```
1: Initialize:  left = 0
2:              right = (count_nodes - 1)
3:              hash =<initial hash value>
4:for i in count_nodes:
5:    total_weight + = EWeights[i]
6:    hash = (hash*33 + IP_Addr[i]) ^(hash ≫ 16)
7:    arr[i] = EWeights[i]
8: QuickSort(arr, 0, 2)
9: w = hash % total_weight
10: while (left < right):
11:  mid=left + (right - left) / 2
12:  w - = arr[mid]
13:       if(w < 0):
14:            p = mid
15:            break
16:       end if
17:       if (w > 0):
18:            left = mid + 1
19:       end if
20:       if (w == 0):
21:            right = mid - 1
22:       end if
23: p = mid
24: SeNode = Nodes[p]
25: Return SeNode
26: free(arr)
```

total_weight, *hash*, and assigning the values of *EWeights[ ]* to the values of *arr* (lines 4-7). And we can do the quicksort and set the value of *hash* (lines 8-9). The next is the binary search in NEW_HASH:

1) Finding the middle index of the array of weights (lines 11) and letting *w* subtracts the value of the middle index position of the array (lines 12). If *w* is less than 0, which satisfies the condition for selecting the back-end node in ALGORITHM 2, we record the current node's index and assign it to *p* (lines 13-16).
2) On the other hand, if w is greater than or equal to 0, move the left or right accordingly until the appropriate position is found and assign it to *p* as the suitable index for the current loop (lines 17-23).

This time, the *SeNode* is the selected node (lines 24-25), and then the algorithm will destroy the *arr* (lines 26) and continue the next loop.

The original principle of binary search is as follows: the element to be looked for is first compared with the element in the middle of the sequence. If it is bigger than this element, the search continues in the second half of the current sequence. If it is less than this element, the search continues in the first half of the current sequence until the same element is found, or the range of the searched sequence is empty. ALGORITHM 5 mentioned in this paper combines the searching process of IP_HASH(in ALGORITHM 2) and binary search to form a more efficient search method for the suitable index of a back-end node. Assuming that N is the length of a sequence of back-end nodes, then the time complexity of sequential search mentioned in IP_HASH is N, denoted as O(N). And the time complexity of binary search mentioned in NEW_HASH is $\log_2(N)$, denoted as $O(\log_2(N))$, so the advantage of this kind of search NEW_HASH can be seen as follows:

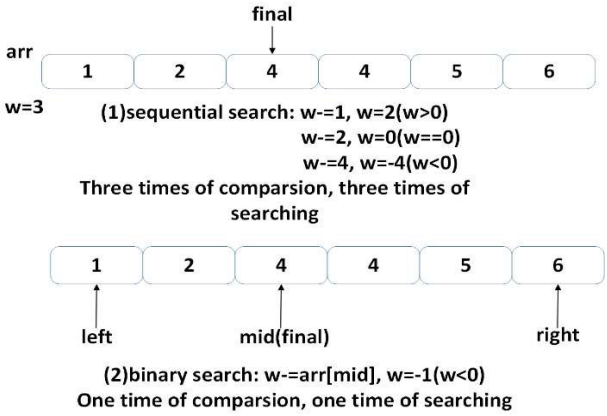


FIGURE 12. The comparison of the binary search and sequential search.

The innovative NEW_HASH algorithm mentioned in this study is introduced in detail in the above analysis. From the two perspectives of the innovation of the hash function and searching way, and by comparing with the old method body of IP_HASH, the new method in NEW_HASH has the characteristics of reducing the possibility of the hash collision, increasing dispersion, and increasing the searching efficiency. So, the NEW_HASH does have advantages in improving load balancing capability from the above study.

IV. THE EVALUATION INDEX OF LOAD BALANCING

In Section 3, we understand the principle and implementation process of NEW_HASH at the code level, and it can be seen that this new algorithm has some innovations. However, before doing experiments to test the algorithm's performance, we need to understand the indicators that affect load balancing in a cluster from the perspective of mathematical models to understand the significance of load balancing in the actual situation more reasonably.

To evaluate load balancing, we must consider timeliness, effectiveness, and comprehensiveness. Today, it is common that server clusters face too much load pressure and different application environments. So, it is more persuasive to use the dynamic performance evaluation to test the performance of a cluster, and just considering static factors is not enough to get a relatively good result.

There are usually three assessment indicators in dynamic performance evaluation:

1. From the perspective of a running project to see the dynamic performance evaluation of every single node.
2. From the perspective of the current load to analyze changes in the weight of every node.
3. From the perspective of the current cluster to calculate the standard deviation of resource usage of a single node and the whole cluster.

A. DYNAMIC PERFORMANCE EVALUATION OF SINGLE NODE

1) THE STATIC PERFORMANCE MODEL C(S_I)

Generally, the static performance factors considered include CPU, memory, disk, and response time. These factors are also parameters of a computational formula of the static performance model. For example, the CPU frequency, memory capacity, disk IO, and network bandwidth are represented in web clusters by P_{cpu}, P_{mem}, P_{disk}, and P_{net}. To facilitate the proper adjustment of the proportion of each parameter for different applications during the system operation, we set a constant coefficient K_i to show the importance of each parameter. And the linear weighting formula of the current service performance $C(S_i)$ made by the above four factors is as follows:

$$C(Si) = K_1 * Pcpu + K_2 * Pmem + K_3 * Pdisk + K_4 * Pnet \tag{1}$$

And $K_1 + K_2 + K_3 + K_4 = 1$. For different applications, the importance of every above parameter is also different. For example, in the environment of classical web projects, proportions of memory and response time are higher; in the environment of some complicated database transactions, importance of the CPU and ram are more prominent. For example, in one typical web cluster, we can set K_1, K_2, K_3, K_4 as 0.15, 0.4, 0.3, and 0.15, respectively, and this means that we regard the memory and disk IO as the influential primary factors. But if the current K_i ratio is not indicative of the current load, the system administrator can update the K_i ratio until it is more convincing in the current environment.

2) THE DYNAMIC PERFORMANCE MODEL L(S_I)

Firstly, we can assume that the number of connections to the current server node is $CONN_i$, connection numbers $CONN_i1$ and $CONN_i2$ are recorded at times T1 and T2. And the running status Sta_1 and Sta_2 of the server are logged. Then the $CONN_i$ between time T1 and T2 is $CONN_iT2$-$CONN_iT1$, and for CPU, we can set a coefficient k1. And k1 = (Sta2(CPU)-Sta1(CPU))/$CONN_i$.

We can quickly determine k2, k3, k4 for memory, disk IO, and network bandwidth. Then R_i = ki/(k1 + k2 + k3 + k4), and the linear formula of the current load performance $L(S_i)$ made by the above factors can be as follows:

$$\begin{aligned} L(Si) = {} & R1 * CONNi * Pcpu + R2 * CONNi * Pmem \\ & + R3 * CONNi * Pdisk + R4 * CONNi * Pnet \end{aligned} \tag{2}$$

The above equation can also be equivalent to as follows:

$$L(Si) = \lambda_1 * Ucpu + \lambda_2 * Umem + \lambda_3 * Udisk + \lambda_4 * Unet \tag{3}$$

And $\lambda 1 + \lambda 2 + \lambda 3 + \lambda 4 = 1$, this indicates that the sum of weight parameters corresponding to each dynamic factor is 1. The CPU utilization, memory utilization, disk utilization, and broadband usage are represented by U_{cpu}, U_{mem}, U_{disk}, and U_{net}. For a single λ_i, its value range represents its proportion in the model and represents its characteristic attributes in the currently considered indicators: free, normal, and overload. After a large number of training samples, the corresponding relationship between the value range of λ_i and its characteristic attributes is shown in the following table:

TABLE 3. The relationship between ranges of λi and characteristic attributes.

	FREE	NORMAL	OVERLOAD
λ_{cpu}	0~0.6	0.6~0.8	0.8~1
λ_{mem}	0~0.6	0.6~0.85	0.85~1
λ_{disk}	0~0.6	0.6~0.85	0.85~1
λ_{net}	0~0.6	0.6~0.85	0.85~1

It can be seen from the above table that when constructing the model, the proportion of dynamic impact factors that need to be taken into account must be kept within a reasonable value range, and this is to ensure the right embodiment of the load capacity in the current cluster environment and the rationality and correctness of the model construction.

B. THE WEIGHT OF EVERY SINGLE NODE

In the actual application environment, the weight of the single server node can be adjusted dynamically by the corresponding algorithm. For standard clusters, every weight of every server node may change with time going and algorithms changing, like the algorithm WRR, which can make clients access the server more randomly. From a mathematical formula point of view, to make user access to the back-end node more stable and durable, we need to reduce jitter times when accessing the cluster. From time t1 to t2, the formula of jitter coefficient d is as follows:

$$d = \left| \frac{L(Si)t0 - L(Si)t1}{\Delta t} \right| \tag{4}$$

And t1-t0 = Δt, $L(S_i)_{t1}$-$L(S_i)_{t0}$ is the difference between the dynamic load indicators of the server node. So a minor d means a more stable cluster and more average weight distribution; more major d means more frequently opening or closing the server node when clients access it, and we can not keep the stability of this cluster for a long time.

The weight formula of single server node S_i is as follows:

$$Weight(Si) = \frac{C(Si) * (1 - L(Si))}{\sum_{i=1}^{n} C(Si)} \tag{5}$$

In equation (5), we can see that if the static performance factors considered of the current cluster do not change, then the static performance model $C(S_i)$ and its sum will not change. And this means that the dynamic performance $L(S_i)$ is the factor that affects the result of the whole equation. Next, we can know that the actual parameters that affect the weight distribution of a single node are the dynamic performance factors and their proportion in model $L(S_i)$, which again proves the importance of selecting ranges of λ_i in equation (3) and maintaining a low jitter in equation (4).

C. RESOURCE USAGE OF THE CURRENT CLUSTER

The state of the current cluster depends on every server node in it. Therefore, the formula of resource usage of every server node is as follows:

$$Ui = \frac{L(Si)}{C(Si)} * 100\% \tag{6}$$

U_i is the resource utilization of the current node, $L(S_i)$ is the load performance of the current node, $C(S_i)$ is the service performance of the current node. The bigger the U_i is, the higher the resource utilization of the current node is, and load balancing is better performed on this node. On the contrary, minor U_i means lower resource utilization. That is to say, there is a problem with factor selection or its proportion selection in equation (3), so its process needs to be reviewed. Therefore, the standard deviation of server resource usage can represent the dispersion degree of the current cluster. The first step is to calculate the average resource usage of the cluster:

$$U_{AVG} = \frac{1}{n}\sum_{i=1}^{n} Ui \tag{7}$$

Then the formula of the standard deviation of resource utilization in the cluster is as follows:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Ui - U_{AVG})^2} \tag{8}$$

In equation (8), the smaller the standard deviation is, the better the load cluster is. On the contrary, the larger its value, the more unbalanced cluster is. Then the load status needs to be reevaluated. Finally, from equation (1) to (8) mentioned above, it can be seen that no matter for dynamic performance, the weight of each node or the resource utilization of the whole cluster, the factors that play a decisive role in them are actually the selection of each dynamic factor and the value range of its proportion.

V. CONSTRUCTION OF EXPERIMENTAL ENVIRONMENT AND RESULT ANALYSIS

A. THE CONSTRUCTION OF THE EXPERIMENTAL ENVIRONMENT

In Section 3, we learned that reducing hash collisions is an advantage of NEW_HASH, but this is only a functional advantage, not an indicator for measuring and comparing all load balancing algorithms. For evaluating and testing the load capacity under the algorithm NEW_HASH, this experiment tested the cluster load under WRR (in ALGORITHM 1), IP_HASH (in ALGORITHM 2), LEAST_CONN (in ALGORITHM 3), and NEW_HASH in sequence. This experimental environment comprises three Tomcat servers, one Nginx server, and one Keepalived server in Linux. The hardware configuration of each server is as the following table:

TABLE 4. The hardware configuration of the experiment environment.

Name	Description
Operation System	Cent OS6 32bit
CPU	1996.249 MHZ; single core
Memory	1GB
Server	Apache Tomcat
Disk	21.5GB
Usable RAM	1028592 kB

This experiment in this paper is tested by using Apache Jmeter, a pressure measuring software designed to evaluate the constructions of client-side and server-side (like web applications). The experiment continued to test the cluster load by setting the number of concurrent threads accessing the server cluster from 200 to 3000 per second and increasing the number of threads by 200 after each test for 5 minutes. The primary data tested were the average response time, access failure rate, and throughput for every different number of concurrent threads within 5 minutes. Different algorithms have different advantages in effect, and also have their own applicable environments and conditions. But certain indicators are needed to test the performance of them. The above three indicators are commonly used to test the performance of different load balancing algorithms. After processing a certain amount of data, the access logging files generated by Tomcat and Nginx are be cleaned up to ensure that the same running state and conditions are provided for each algorithm as much as possible under control. In addition, the whole experiment will be carried out under the condition of a good network, and the inconsistency and unreliability of data caused by network lag or latency problems will not occur.

B. RESULT ANALYSIS

Fig. 13 shows that when the number of threads per second is negligible, except WRR, the difference between the average response time of IP_HASH and LEAST_CONN is not significant, and the average response time of NEW_HASH is the least of all. However, when the number of threads is 2800, the advantage of NEW_HASH is more prominent, and it is the best of all algorithms tested. Here is the average response time expression in Jmeter:

$$R = Tsum/Nsum \tag{9}$$

Response time is how long it takes for the application server to return the request result to clients. It is affected by

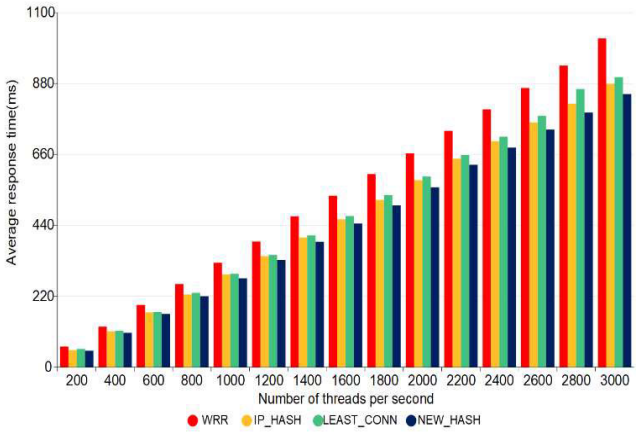


FIGURE 13. Average response time comparison of different algorithms.

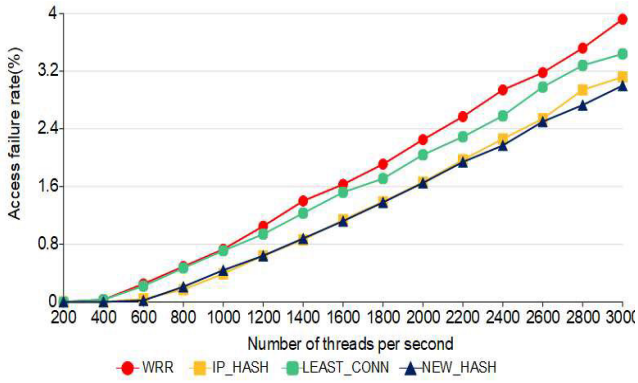


FIGURE 14. Access failure rate comparison of different algorithms.

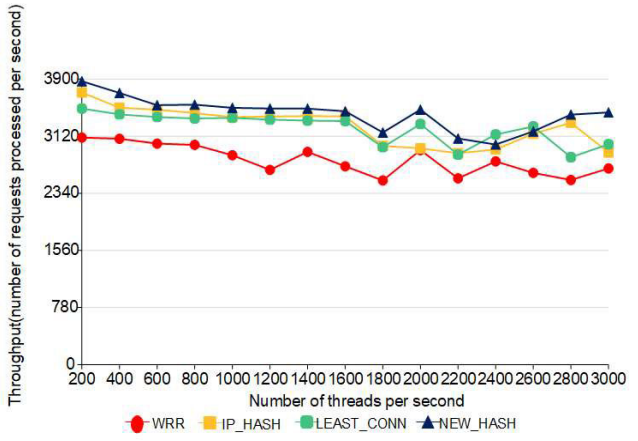


FIGURE 15. Throughput comparison of the four algorithms.

many factors, such as network bandwidth and type of requests submitted by the client, and average latency. Therefore, it is necessary to ensure the consistency of the experimental environment before testing its value. However, when evaluating the loading performance of the cluster, the average time of all requests is the yardstick for analysis. In equation (9), R is the average response time, T_{sum} is the sum of response times of requests, and N_{sum} is the total number of requests. For NEW_HASH, when the number of requests remains unchanged, T_{sum} under a large number of requests becomes smaller due to the increased query performance, resulting in a smaller R. For the first three native methods of Nginx, IP_HASH, compared with WRR and LEAST_CONN, has the shortest average response time as the number of threads per second increases, because the number of loops and selections in IP_HASH is small. On this basis, NEW_HASH improves access efficiency and reduces average response time by changing the hash function and searching way when the number of client access increases. So, the above mentioned shows that the NEW_HASH has the fastest response time among the four algorithms and also increases the performance of the entire load cluster to process access times.

Fig. 14 shows that when the number of threads per second is small, the access failure rate of all four algorithms remains below 0 to 0.8. However, when the number of threads per second increases, the access failure rate of WRR and LEAST_CONN is much higher than that of IP_HASH and NEW_HASH. The access failure rate of NEW_HASH is the lowest among the four algorithms, and its curve increases steadily. Here is the access failure rate expression in Jmeter:

$$F = (Wsum * 100)/Nsum \tag{10}$$

In equation (10), F is the access failure rate, W_{sum} is the number of failed requests, and N_{sum} is the total number of requests. For the four algorithms in Fig. 14, WRR and LEAST_CONN have more times of locating and comparing access nodes than the latter two algorithms. And this will lead to a phenomenon that locating and accessing the node in the current request is not completed, the next request will start soon. Especially when many requests access a load cluster at the same time in a certain time, due to the limitations of the server itself to handle concurrency, the access failure rate is often too high. However, for the NEW_HASH, the time to locate the back-end node is greatly decreased due to the advantage of its searching algorithm. So, the above mentioned shows that when the number of threads per second increases, the NEW_HASH can provide the current server cluster with a better access success rate and access stability.

Fig. 15 shows that among the first three algorithms, WRR has the lowest throughput. When the number of threads per second is low, the throughput of IP_HASH is slightly higher than that of LEAST_CONN. But when the number of threads per second increases, the throughput of LEAST_CONN gradually overtakes the throughput of IP_HASH. The throughput of IP_HASH and NEW_HASH is lower than that of LEAST_CONN when the number of threads per second is 2600. However, the throughput of NEW_HASH gradually exceeds that of LEAST_CONN when the number of threads per second increases. Here is the throughput expression in Jmeter:

$$T = TPS = (Nnum)/(Tsum) \tag{11}$$

In equation (11), T is the throughput, TPS is transactions processed per second, N_{num} is the total number of requests, and T_{sum} is the sum of response times of requests.

This formula could have been written in the following form:

$$T = N\text{sum}/(T_{last\_start} + T_{last\_duration} - T_{first\_start}) \tag{12}$$

T_{last_start} is when the last thread starts, T_{first_start} is when the first thread starts, and $T_{last\_duration}$ is the duration of the last thread. And then, we can see from equation (12) and Fig. 15 that the total concurrency does not exist, and requests are always sent in sequential order. This is why we need to keep increasing the number of requests to find the maximum number of TPS.

For the four algorithms, when the number of threads per second increases for a while, the corresponding throughput will sometimes increase or decrease. In Fig. 15, when the number of threads is 2000 to 2600, the throughput of NEW_HASH will get a heavy decrease. That is, the increase of the curve is not very stable. The search performance of binary search is improved. Still, due to the existence of array creation, elimination, and quicksort, as the number of concurrent requests increases, the overhead of the server will be mainly distributed to the creation and elimination of space, resulting in a decrease in throughput or even an unstable situation. It shows that although the speed and performance of this algorithm are improved, it is not stable. The server load and memory pressure need to be further studied and improved.

C. THE ANALYSIS OF DIFFERENT NUMBERS OF SERVER NODES

According to the above results in section B, we know that this new algorithm's query speed, accuracy, and throughput have been improved in the same cluster. And this means that NEW_HASH is actually better than the other three algorithms when dealing with large number of highly concurrent requests.

After comparing different algorithms, we need to analyze further to compare NEW_HASH itself under different application conditions. For example, how well does it perform when faced with different numbers of service nodes? Let us do the theoretical analysis firstly:

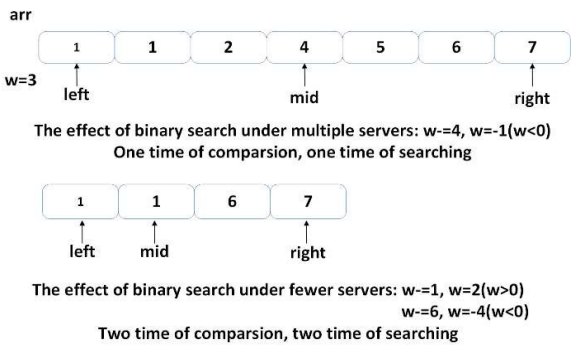


FIGURE 16. The analysis of the theory.

In the case shown in Fig. 16, traversal times and searching times may increase after reducing some server nodes. That is to say that the algorithm can theoretically increase the efficiency and performance of load balancing for the partial configuration environment of multiple clusters. In a more complex multi-node business environment, the load pressure can be relieved as much as possible. The following three figures show the comparison of average response time, access failure rate, and throughput under different numbers of nodes:

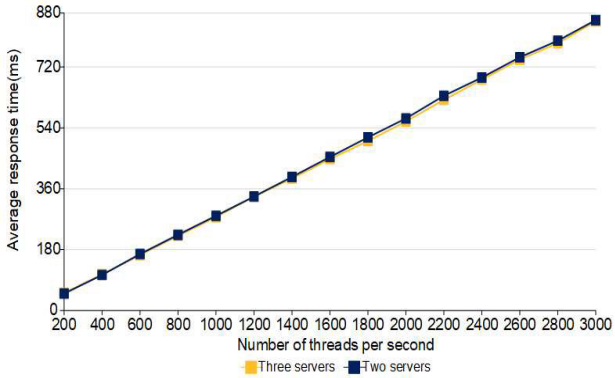


FIGURE 17. Average response time comparison in different cluster environments.

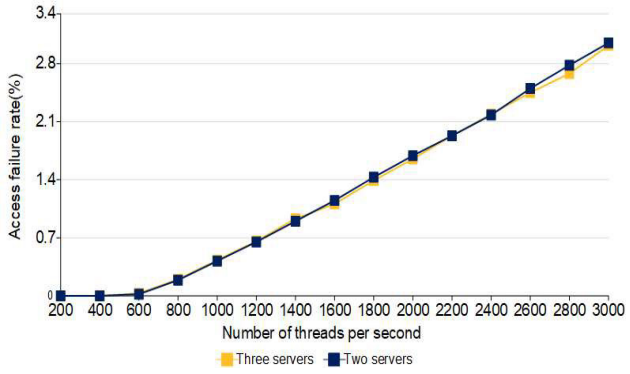


FIGURE 18. Access failure rate comparison in different cluster environments.

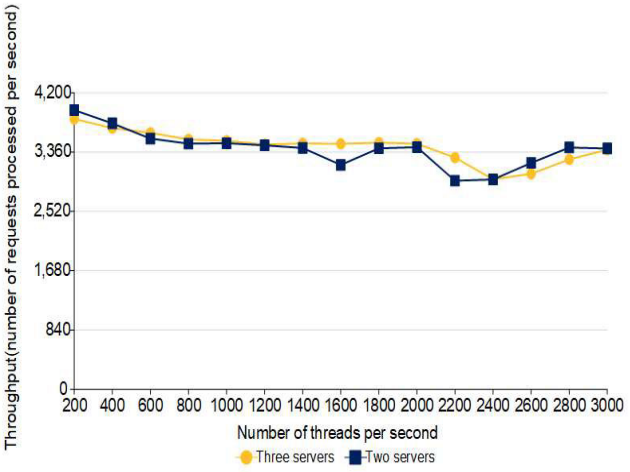


FIGURE 19. Throughput comparison in different cluster environments.

In Fig. 17, it can be seen that the curve of the average response time of three server nodes increases by about the same amount as that of two server nodes, and after the number

of threads per second reaches 1800, the average response time on three server nodes drops slightly. These details show that this new algorithm can keep the average response time basically unchanged in the environments with little difference in the number of server nodes and reflect the stability of the searching performance. In Fig. 18, the difference between the curves of access failure rates is also tiny, and when the number of threads per second increases to 2600, the failure rate of three server nodes is slightly less than that of two server nodes. To sum up, this algorithm also ensures the success rate of clients accessing the cluster environments with little difference in the number of server nodes.

In Fig. 19, we can see that there is little difference between the curve of three server nodes and two server nodes when the number of threads per second is small. But after the number of threads per second reaches 1400, there are some points in the figure where the throughput of three server nodes is higher than that of two server nodes. These changes explain that when there are more server nodes in a server cluster because the hash function included in NEW_HASH reduces hash collisions, the number of requests is distributed more evenly to each server node. As a result, the throughput on each node increases, and the load pressure on each node decreases per unit time. Finally, when the number of requests reaches a certain number, due to the creation, destruction of array space, and quicksort, many requests are not processed in a unit time, resulting in a decrease in throughput.

D. THE ANALYSIS OF HASH COLLISION REDUCTION

As can be seen from section B and C above, the NEW_HASH has relatively good load balancing effect for different cluster environment based on data analysis of average response time, access failure rate and throughput. From Fig. 9, we can conclude that one of the advantages of this algorithm is to reduce hash collisions. So how do we prove this feature of reducing hash collisions experimentally? The steps can be as follows:

1) First, we set up different IP address on the ethernet adapter on the current load machine, and also set up the default gateway and DNS server address associated with it.
2) Then, we set up corresponding operations on Jmeter, assigning the above corresponding different IP addresses to the parameters of Jmeter's requested address in turn. In this way, we can use Jmeter to simulate different IP addresses to send requests to the current cluster so that we can see which back-end node is visited. Finally, we will see if the NEW_HASH really has the advantage of reducing hash collisions.

In this experiment, we randomly generated 1000 IP addresses by using a java program to test the effect of hash collisions under the load environment of IP_HASH and NEW_HASH respectively, with three servers in both tests. Then we repeated the experiment 5 times and recorded the times of each server node being accessed each time. The mapping between visits and server nodes is as follows:

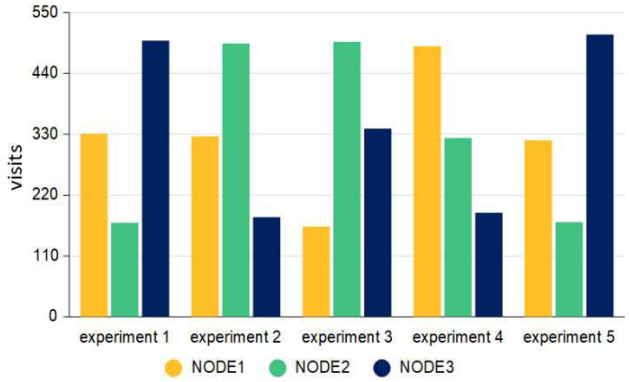


FIGURE 20. The mapping in IP_HASH.

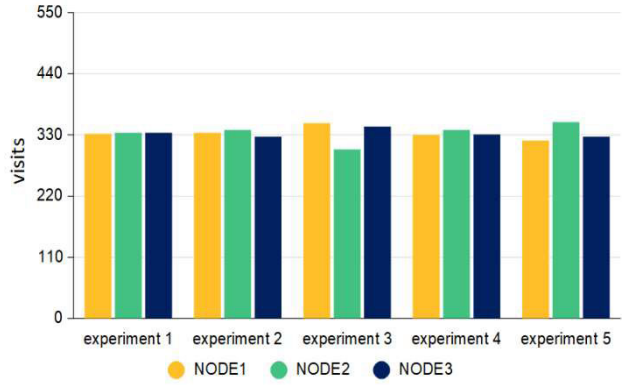


FIGURE 21. The mapping in NEW_HASH.

As we can see from Fig. 20, the difference among the number of visits on the three service nodes is excessive. We can see that in each experiment, there is always a server node with too many visits, which will lead to the increase of the load pressure on the server. In addition, there is always a server node with too small visits, which leads to insufficient resource utilization of the node and affects the load efficiency of the entire load cluster. As we know from the above, the probability of hash collisions caused by IP_HASH is relatively high.

But in Fig. 21, we can see that the number of accesses on each server node is relatively average. In each experiment, the difference among the number of visits at each node is small, which led to the relatively normal load pressure at each node. In this way, there is no insufficient or high resource utilization on a node, which improves load balancing. It is clear that NEW_HASH does reduce the probability of hash collisions, and in the actual application environment, this algorithm will also significantly reduce the memory consumption of the back-end cluster environment.

VI. CONCLUSION

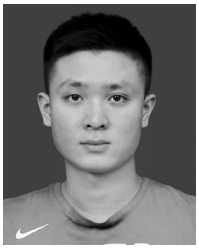
This paper proposes a load balancing algorithm NEW_HASH to reduce the hash collision in a whole cluster and improve the performance of search nodes. The IP_HASH algorithm in Nginx suffers from high collision rate and poor search performance. The NEW_HASH algorithm improves its hash function and searching method, making its concept richer in

theory and logic more detailed in code. As a result, compared with the three native Nginx algorithms, it has significant advantages in reducing the average response time, access failure rate, and improving throughput. In general, this new algorithm can improve the performance of load balancing and better reduce the load pressure of the back-end cluster.

However, this algorithm also has disadvantages. For example, when the back-end cluster is traversed, an additional array is created to store the respective weights of the back-end nodes, and it needs to be destroyed when selecting back-end nodes. Such operations will increase the space overhead of the whole cluster, which reduces its searching performance and efficiency when dealing with large concurrent requests. Therefore, future research should focus on reducing the space overhead of the load balancing algorithm and establish more persuasive models to increase the reliability of the load balancing algorithm in the actual business environment.

REFERENCES

[1] L.-H. Hung, C.-H. Wu, C.-H. Tsai, and H.-C. Huang, "Migration-based load balance of virtual machine servers in cloud computing by load prediction using genetic-based methods," *IEEE Access*, vol. 9, pp. 49760–49773, 2021.
[2] Y. Cao, "Load balancing design of web cluster based on nginx under novel virtualization platform," in *Proc. Int. Conf. Comput. Commun. Artif. Intell. (CCAI)*, May 2021, pp. 166–170.
[3] N. Sun, H. Z. Shi, G. J. Han, B. Wang, and L. Shu, "Dynamic path planning algorithms with load balancing based on data prediction for smart transportation systems," *IEEE Access*, vol. 8, pp. 15907–15922, 2020.
[4] L. Zhu, J. Cui, and G. Xiong, "Improved dynamic load balancing algorithm based on least-connection scheduling," in *Proc. IEEE 4th Inf. Technol. Mechatronics Eng. Conf. (ITOEC)*, Dec. 2018, pp. 1858–1862.
[5] B. Li, J. Shang, M. Dong, and Y. He, "Research and application of server cluster load balancing technology," in *Proc. IEEE 4th Inf. Technol., Netw., Electron. Autom. Control Conf. (ITNEC)*, Jun. 2020, pp. 2622–2625.
[6] X. Chi, B. Liu, Q. Niu, and Q. Wu, "Web load balance and cache optimization design based Nginx under high-concurrency environment," in *Proc. 3rd Int. Conf. Digit. Manuf. Autom.*, Jul. 2012, pp. 1029–1032.
[7] L. H. Pramono, R. C. Buwono, and Y. G. Waskito, "Round-robin algorithm in HAProxy and Nginx load balancing performance evaluation: A review," in *Proc. Int. Seminar Res. Inf. Technol. Intell. Syst. (ISRITI)*, Nov. 2018, pp. 367–372.
[8] D. A. Shafiq, N. Z. Jhanjhi, A. Abdullah, and M. A. Alzain, "A load balancing algorithm for the data centres to optimize cloud computing applications," *IEEE Access*, vol. 9, pp. 41731–41744, 2021.
[9] I. Ivanisenko and M. Volk, "Simulation methods for load balancing in distributed computing," in *Proc. IEEE East-West Design Test Symp. (EWDTS)*, Sep. 2017, pp. 1–6.
[10] K. Kaur and A. Kaur, "A hybrid approach of load balancing through VMs using ACO, MinMax and genetic algorithm," in *Proc. 2nd Int. Conf. Next Gener. Comput. Technol. (NGCT)*, Dehradun, India, Oct. 2016, pp. 615–620.
[11] Y. Elshater, P. Martin, and E. Hassanein, "Using design patterns to improve web service performance," in *Proc. IEEE Int. Conf. Services Comput.*, New York, NY, USA, Jun. 2015, pp. 746–749.
[12] O. Adekoya, A. Aneiba, and M. Patwary, "An improved switch migration decision algorithm for SDN load balancing," *IEEE Open J. Commun. Soc.*, vol. 1, pp. 1602–1613, 2020.
[13] R. Khan, "An efficient load balancing and performance optimization scheme for constraint oriented networks," *Simul. Model. Pract. Theory*, vol. 96, Nov. 2019, Art. no. 101930.
[14] T. Hu, P. Yi, J. Zhang, and J. Lan, "Reliable and load balance-aware multi-controller deployment in SDN," *China Commun.*, vol. 15, no. 11, pp. 184–198, Nov. 2018.
[15] F. Yan, X. Xue, X. Guo, B. Pan, J. Wang, S. Zhang, E. Khani, G. Guelbenzu, and N. Calabretta, "Load balance algorithm for an OPSquare datacenter network under real application traffic," *J. Opt. Commun. Netw.*, vol. 12, no. 8, pp. 239–250, Aug. 2020.
[16] A. Hava, Y. G. Doudane, J. Murphy, and G. M. Muntean, "A load balancing solution for improving video quality in loaded wireless network conditions," *IEEE Trans. Broadcasting*, vol. 65, no. 4, pp. 742–754, Dec. 2019.
[17] Y. Zhao, X. Wang, Q. He, C. Zhang, and M. Huang, "PLOFR: An online flow route framework for power saving and load balance in SDN," *IEEE Syst. J.*, vol. 15, no. 1, pp. 526–537, Mar. 2021.
[18] X. Wu and H. Haas, "Load balancing for hybrid LiFi and WiFi networks: To tackle user mobility and light-path blockage," *IEEE Trans. Commun.*, vol. 68, no. 3, pp. 1675–1683, Mar. 2020.
[19] J. Liu, R. Luo, T. Huang, and C. Meng, "A load balancing routing strategy for LEO satellite network," *IEEE Access*, vol. 8, pp. 155136–155144, 2020.
[20] Y.-C. Wang and S.-Y. You, "An efficient route management framework for load balance and overhead reduction in SDN-based data center networks," *IEEE Trans. Netw. Service Manage.*, vol. 15, no. 4, pp. 1422–1434, Dec. 2018.
[21] M. C. Lucas-Estañ and J. Gozalvez, "Load balancing for reliable self-organizing industrial IoT networks," *IEEE Trans. Ind. Informat.*, vol. 15, no. 9, pp. 5052–5063, Sep. 2019.
[22] P. H. Pednekar, W. Hallberg, C. Fager, and T. W. Barton, "Analysis and design of a Doherty-like RF-input load modulated balanced amplifier," *IEEE Trans. Microw. Theory Techn.*, vol. 66, no. 12, pp. 5322–5335, Dec. 2018.
[23] D. Marabissi, G. Bartoli, and A. Stomaci, "Low-complexity distributed cell-specific bias calculation for load balancing in UDNs," *IEEE Trans. Veh. Technol.*, vol. 68, no. 1, pp. 1056–1060, Jan. 2019.
[24] L. Yang, H. Yao, J. Wang, C. Jiang, A. Benslimane, and Y. Liu, "Multi-UAV-enabled load-balance mobile-edge computing for IoT networks," *IEEE Internet Things J.*, vol. 7, no. 8, pp. 6898–6908, Aug. 2020.



CHEN MA received the Bachelor of Engineering degree from Shanxi University, Taiyuan, China, in 2021. He is currently pursuing the master's degree in electronic information with Xijing University, Xi'an, Shaanxi. His research interests include big data analysis and software engineering.

YUHONG CHI received the master's degree in computer applied engineering from Northeastern University, Liaoning, China, in 2005, and the Ph.D. degree in computer science and technology from Tsinghua University, Beijing, China, in 2013. Her research interests include computational intelligence and its applications and data analysis.