Computer Systems Science & Engineering
DOI:10.32604/csse.2021.016683
Article

Tech Science Press

Exploration on the Load Balancing Technique for Platform of Internet of Things

Donglei Lu[1], Dongjie Zhu[2,*], Yundong Sun[3], Haiwen Du[3], Xiaofang Li[4], Rongning Qu[4], Yansong Wang[3], Ning Cao[1] and Helen Min Zhou[5]

[1]School of Artificial Intelligence, Wuxi Vocational College of Science and Technology, Wuxi, 214028, China
[2]School of Computer Science and Technology, Harbin Institute of Technology, Weihai, 264209, China
[3]School of Astronautics, Harbin Institute of Technology, Harbin, 150001, China
[4]Department of Mathematics, Harbin Institute of Technology, Weihai, 264209, China
[5]School of Engineering, Manukau Institute of Technology, Auckland, 2241, New Zealand
*Corresponding Author: Dongjie Zhu. Email: zhudongjie@hit.edu.cn
Received: 08 January 2021; Accepted: 27 February 2021

Abstract: In recent years, the Internet of Things technology has developed rapidly, and smart Internet of Things devices have also been widely popularized. A large amount of data is generated every moment. Now we are in the era of big data in the Internet of Things. The rapid growth of massive data has brought great challenges to storage technology, which cannot be well coped with by traditional storage technology. The demand for massive data storage has given birth to cloud storage technology. Load balancing technology plays an important role in improving the performance and resource utilization of cloud storage systems. Therefore, it is of great practical significance to study how to improve the performance and resource utilization of cloud storage systems through load balancing technology. On the basis of studying the read strategy of Swift, this article proposes a reread strategy based on load balancing of storage resources to solve the problem of unbalanced read load between interruptions caused by random data copying in Swift. The storage asynchronously tracks the I/O conversion to select the storage with the smallest load for asynchronous reading. The experimental results indicate that the proposed strategy can achieve a better load balancing state in terms of storage I/O utilization and CPU utilization than the random read strategy index of Swift.

Keywords: The Internet of Things; cloud storage; Swift; load balancing; scheduling algorithm

1 Introduction

With the rapid development of the Internet of Things technology and the popularization of smart mobile terminal devices, massive amounts of data are being generated at all times, marking that it has now entered the era of big data [1,2], and posing a huge challenge to storage systems. However, traditional storage technology cannot respond well to the demand for massive data storage, cloud storage technology thus has emerged at the historic moment [3,4]. At present, there are many new cloud storage systems with high scalability such as Swift [5,6]. Swift, an object storage sub-project of OpenStack, provides cloud

storage services that store and retrieve large amounts of data through simple RESTful APIs. Its goal is to provide data storage with high scalability, high availability, and high durability [7,8].

As a cloud storage system, Swift can better meet the needs of massive data storage capacity, but its system performance and resource utilization still need to be urgently optimized. In Swift, when a user requests to download an object, the proxy server first finds the storage queue where all copies of the object are located from the object ring according to the consistent hash algorithm. Then it uses an algorithm to randomly sort these binaries, and finally reads the data on the storage array according to the sorted order. It is possible to assign many read requests to a single storage system because it is random without considering the difference in the performance of the storage subsystems, making the response speed of the system directly affected by busy equipment and other problems. The storage routine of the copy of the object may have a relatively large upper limit, resulting in a load imbalance. When the load is unbalanced, it will cause a waste of system resources, and also affect the response time and throughput of the whole system, resulting in poor user experience. To solve this problem, a better method is needed to allocate read requests, reasonably use the resources of the cloud storage system, achieve load balancing during reading, and improve the availability of the entire cloud storage system.

Based on the above analysis, this paper proposes a read strategy based on load balancing of storage node resources. Under this strategy, the proxy node selects the storage node with the least load for reading based on the I/O utilization rate received from the storage node in real-time. The experimental results show that this strategy can achieve a better load balancing state in terms of storage I/O utilization and CPU utilization than the random read strategy index of Swift.

2 Related Work

The research progress of load balancing technology is described in this section. Load balancing is mainly used to expand the bandwidth of the system, improve its resouce utilization, reduce its response time, avoid a single point of failure in the system, and enable users to obtain relatively consistent access quality no matter where they are [9].

DNS load balancing is the first load balancing technology used at present. The same domain name corresponds to multiple IPs on the domain name server. When a user initiates a request, the domain name will be resolved by the domain name server to a different IP address, and the user request will also be distributed to different servers for processing [10]. The DNS load balancing scheme is relatively simple to operate, but it also has some disadvantages. First, the polling algorithm is used for resolving the domain name to the server, and load distribution cannot be performed based on the difference in server processing capabilities. In this case, the worst-performing server in the cluster will become the bottleneck of the system, and the server with strong processing capability will not function well. Second, when a host server is unavailable due to a failure, user requests will still be distributed to the server, but the server cannot respond to user requests. At this time, the server needs to be removed from the DNS settings. The modified settings will not take effect after a certain period of time because there are multiple domain name servers between the user and the domain name server, and there are caches on the domain name servers. During this period, none of the requests assigned to the server can be responded to.

Many researchers have proposed improvements to load balancing scheduling algorithms for specific application scenarios and different goals in their research on load balancing scheduling algorithms. Sun et al. believe that the traditional static load balancing scheduling algorithm may cause a large difference in the CPU utilization of the backend server. According to the CPU and memory usage of the server, they proposed a genetic algorithm-based load balancing scheduling algorithm [11]. The difference in CPU utilization of the back-end servers in this algorithm is less than 35%. To overcome the shortcomings of the traditional genetic algorithms such as local convergence and premature, Yang et al. [12] proposed a

load balancing scheduling algorithm based on an adaptive niche genetic algorithm to reduce the average response time of the system. Li et al. [13] proposed a load balancing scheduling algorithm based on the ant colony algorithm to achieve the load balancing of cloud computing resources through the adaptive heuristic information of ant colony. Aiming at the deficiencies of the traditional ant colony algorithm such as single pheromone, single population, prone to premature stagnation, and slow convergence, Shi et al. [14] proposed a load balancing algorithm based on an improved polymorphic ant colony by adding elite ant pheromone and local optimization pheromone based on traditional ant colony algorithm and using local detection and global search methods to make different types of ant colonies work together. Hu et al. [15] proposed a consistent hash load balancing scheduling algorithm based on dynamic feedback by combining the dynamic feedback mechanism and the consistent hashing algorithm to achieve the real-time adjustment of the server load so that the system could achieve a load balancing state. Wang et al. [16] proposed a grid workflow task scheduling algorithm based on load balancing, which focused on the prediction of node load weights, mainly for the optimization of grid system. Tan et al. [17] proposed to evaluate the load of the server using the group decision analytic hierarchy process while establishing a load prediction model using a neural network and then combined the weighted round-robin algorithm with the load prediction model to propose a dynamic load balancing scheduling algorithm. The algorithm calculates and updates the weight of the server according to the results of load prediction. In terms of the scheduling time span and resource load balance, Lu et al. [18] proposed a trust-driven resource load balancing scheduling algorithm to improve the load balance of system resources. Wang et al. [19] proposed a dynamic load balancing algorithm based on the division of task type. User requests are divided into CPU consumption and I/O consumption according to their resource consumption. The load balancing scheduler allocates load according to the request type to improve system throughput.

3 Load Balancing Read Strategy Based on Swift Storage Node Resources

3.1 Strategy Design

In Swift, to achieve reliable data storage, the same data will be stored in three copies by default, and to avoid data loss caused by a failure of a storage node, three copies of the data will be stored separately in different areas. Here, the area is a logical concept, which can be a rack or a data center. Therefore, different copies will not be stored on the same storage node but will be stored on different storage nodes. Swift's read strategy is to randomly select a storage node to read after obtaining all the storage nodes of the data copy. This is likely to result in some storage nodes reading with too many requests, while others are relatively idle, resulting in an unbalanced load. The key to the improvement strategy proposed in this paper is to make use of the characteristics of replicas on different storage nodes to allocate read requests more reasonably, so as to achieve the purpose of rational use of resources and load balancing.

The improvement strategy proposed in this paper mainly includes load information collection module and sorting module.

(1) Load information collection module.

The main function of the load information collection module is to collect the load information of the storage node. When a user makes requests, the proxy node will first accept the user's request, then find the specific storage node where the user's request object is located according to the consistent hash ring, and finally submit the request to the storage node for processing. Therefore, the specific processing of data is implemented on the storage node. The specific operations of data mainly include data storage and data reading, and these operations are inseparable from the disk. When the disk is idle, data reading is faster, and vice versa. Therefore, this paper proposes indicators based on I/O substitution. The storage stack collects its own disk I/O at regular intervals and feeds it back to the proxy agent.

(2) Sorting module.

There are multiple copies of data in Swift. When a user initiates a download request, the specific storage node to access to read is selected by the proxy node. In the reading strategy proposed in this paper, the proxy server selects the storage routine with the smallest disk I/O for reading. The proxy node then maintains a list of the IP addresses of the storage node and the disk I/O utilization of the storage node. When the disk I/O utilization rate is received from the storage node, the list will update the corresponding the Disk I/O utilization of the storage node. When the proxy node receives the download request from the user, it obtains the storage node where all copies of the object are located by querying the consistent hash ring. The storage node where the object copy is located is then sorted according to the disk I/O utilization in the storage node information list maintained by the agent node. Finally, the data is read according to the sorted storage node order. If the reading is successful, the result is directly returned, if it fails, the data is read from the second storage node in the list. And the rest can be done in the same manner. The reading of the improvement strategy is shown in Fig. 1.

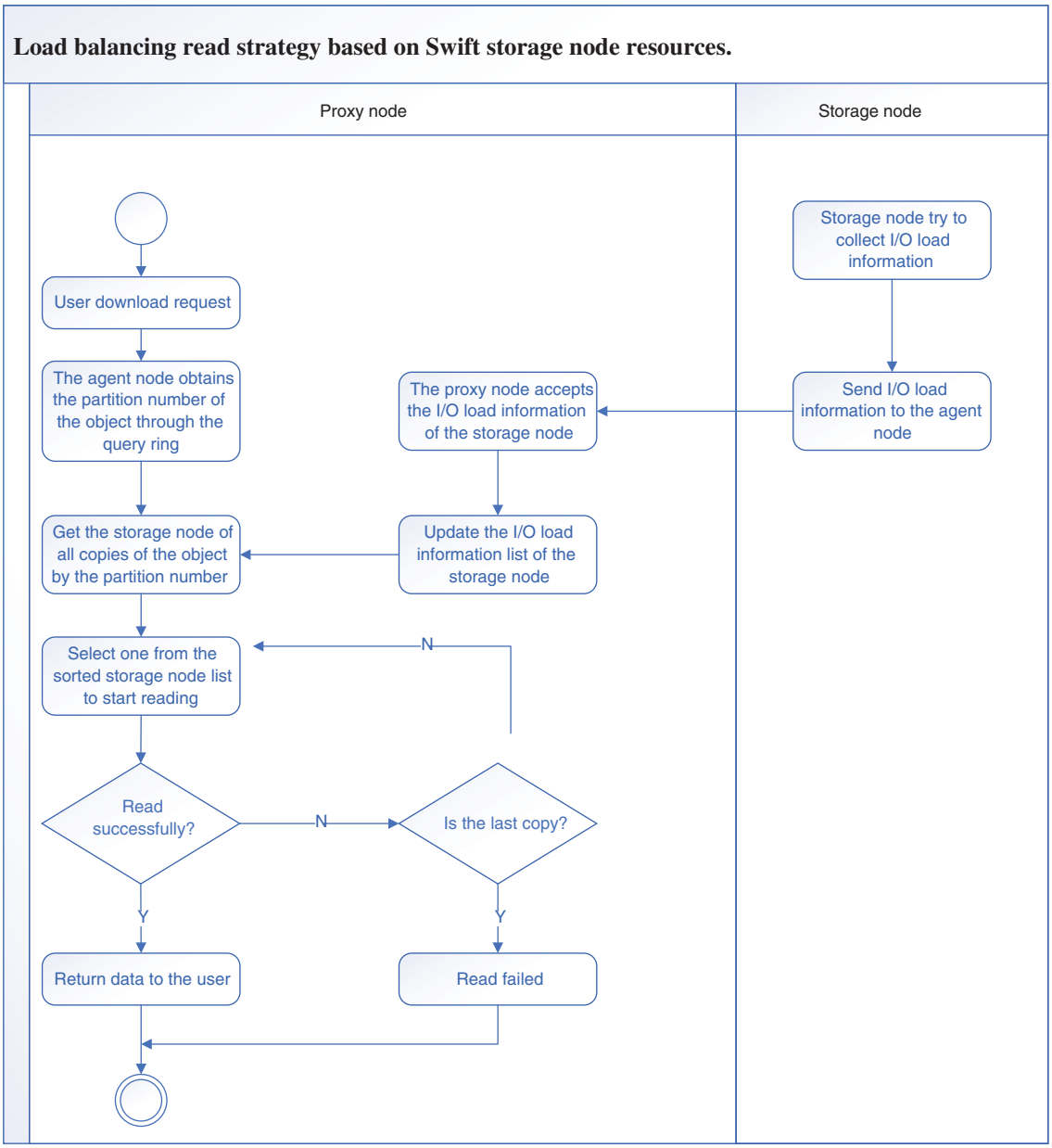

Figure 1: Diagram of reading strategy based on storage node resource load balancing

3.2 Strategy Implementation

The implementation of the Swift reading strategy based on the load balancing of storage node resources mainly includes an I/O load information collection module, a communication module, and a sorting module. The relationship of each module is shown in Fig. 2.

Figure 2: The relationship between the modules of the improved read strategy

(1) Load collection module.

The load information in this paper refers to the I/O utilization of disk. Under Linux, the iostat command is used to monitor the I/O status of the system, and the disk utilization can be viewed through the iostat-x command, as shown in Fig. 3.

```
swift@proxy1:~$ iostat -x 1 1
Linux 3.11.0-15-generic (proxy1)        05/18/2015      _x86_64_        (2 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           3.96    0.01    0.89    0.97    0.00   94.17

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.29     0.35    3.10    0.36   279.41    10.41   167.49     0.08   23.85    9.12  151.86   5.01   1.73
```

Figure 3: I/O usage

where %util represents the ratio of all processing I/O time to the total statistical time during the statistical time. For example, if the statistical time is 1 second, and 0.0173 seconds are processing I/O in the above figure, then %util = 0.0173/1 = 1.73%. The larger the %util, the busier the disk.

(2) Communication module.

The communication module includes server and client, and the client runs on the storage node, collects its own disk utilization in real time, and then calls the SendLoad (IP, load) method to send its own disk utilization to the server. The server runs on the agent node and maintains a storage Node List of storage node information. The list contains IP and load fields. The server has registered the SendLoad IP, load) method. When the client calls this method, the load information of the corresponding IP in the storage Node List will be updated. The server also registers a GetLoad() method, which can be called to get storage node information storage Node List.

(3) Sorting module

According to the download request of the user, the proxy node obtains a list of nodes of all storage nodes of a copy of the requested data by searching the ring. The number of storage nodes included in this list is the number of copies. The structure diagram of the storage node is shown (this diagram is above), which contains the IP information of the storage node. The storage node information storage Node List maintained by the agent node can be obtained by calling the GetLoad() method. Then the storage node list nodes are sorted in an ascending order according to the load size of the storage node in the storage Node List, so that the storage node with a small load is sorted to the front of the list, and the storage node with a large load is sorted to the back of the list. Then the system will prioritize the allocation of read requests to the storage node with the least load.

4 Experiment and Result Analysis

4.1 Experimental Environment Construction

According to the three-copy storage strategy advocated in Swift, the experimental deployment includes authentication nodes, proxy nodes, storage nodes, and test clients. The experimental environment is shown in Tab. 1.

Table 1: Experiment environment

Machine IP	Deployment service	Description
172.29.132.51	Object Server, Container Server, Accoount Server	Storage node
172.29.132.52	Object Server, Container Server, Accoount Server	Storage node
172.29.132.53	Keystone, Proxy Server, Object Server, Container Server, Accoount Server	Authentication nodeProxy node, Storage node
172.29.132.55	Test client	Test client

4.2 Experimental Results and Analysis

In the Swift cloud storage system, I/O utilization, CPU utilization, and network outflow rate can well represent the resource load status of storage nodes. In this paper, I/O utilization, CPU utilization, and network outflow rate are used to compare the resource load balance of storage nodes in Swift's read strategy and the improved read strategy.

The experimental scenario is to read Swift itself after randomly sorting the storage nodes in the replication list, and to read the storage nodes in the replication list after sorting according to the I/O load conditions proposed in this paper

In the experiment, the number of concurrent users was 5, 10, 15, 20, 30, 40, and 50, and the file size downloaded by users was 30M and 50M, respectively. The number of users increased gradually, with an increase interval of 2s and 3s, respectively. In the strategy proposed in this paper, the I/O load information of the storage node was collected every two seconds and the load information was reported to the agent node. It was used by the proxy node to select the storage node from which to read the replica for reading. The following table shows the time required for downloading (unit: s):

Taking 50 concurrent users as an example, the Swift read strategy proposed in this paper was compared with the read strategy based on load balancing of storage node resources under the scenario of downloading files of different sizes and time intervals, I/O utilization, CPU utilization, and network outflow rates.

A file with a size of 30M was downloaded, the number of users was increased every two seconds till there were 50 users. According to the I/O load proposed in this paper, the I/O utilization, CPU utilization, speed and network outflow rate of storage nodes are shown in Fig. 4.

A file with a size of 30 M was downloaded, the number of users was increased every three seconds till there were 50 users. According to the I/O load proposed in this paper, the I/O utilization, CPU utilization, speed and network outflow rate of storage nodes are shown in Fig. 5.

A file with a size of 50 M was downloaded, the number of users was increased every two seconds till there were 50 users. According to the I/O load proposed in this paper, the I/O utilization, CPU utilization, speed and network outflow rate of storage nodes are shown in Fig. 6.

A file with a size of 50M was downloaded, the number of users was increased every three seconds till there were 50 users. According to the I/O load proposed in this paper, the I/O utilization, CPU utilization, speed and network outflow rate of storage nodes are shown in Fig. 7.

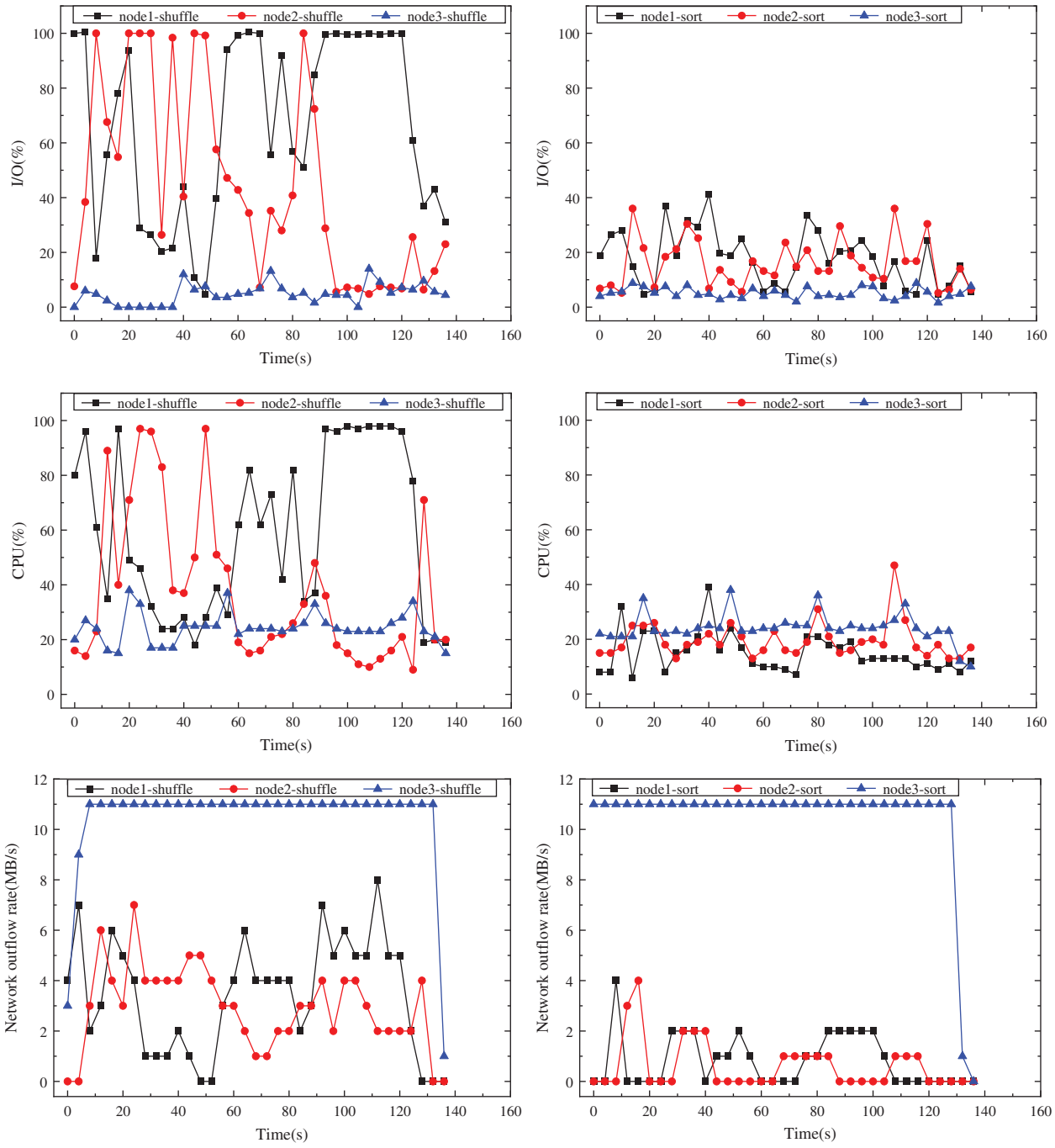

Figure 4: I/O utilization, CPU utilization and network outflow rate at 30M/2s

According to the above experimental results, if a storage node was randomly selected for reading in Swift, some storage node I/O utilization and CPU utilization were too high, while some other storage node I/O utilization and CPU utilization rate were relatively low, resulting in a waste of resources. The resource load balancing read strategy based on I/O utilization proposed in this paper will not cause the I/O utilization and CPU utilization of some storage nodes to be too high, while the I/O utilization and CPU utilization of other storage nodes When the rate is too low, the I/O utilization and CPU utilization of

each storage node is relatively close. It shows that the resource load balancing read strategy based on I/O utilization has a good effect on I/O resource and CPU resource load balancing.

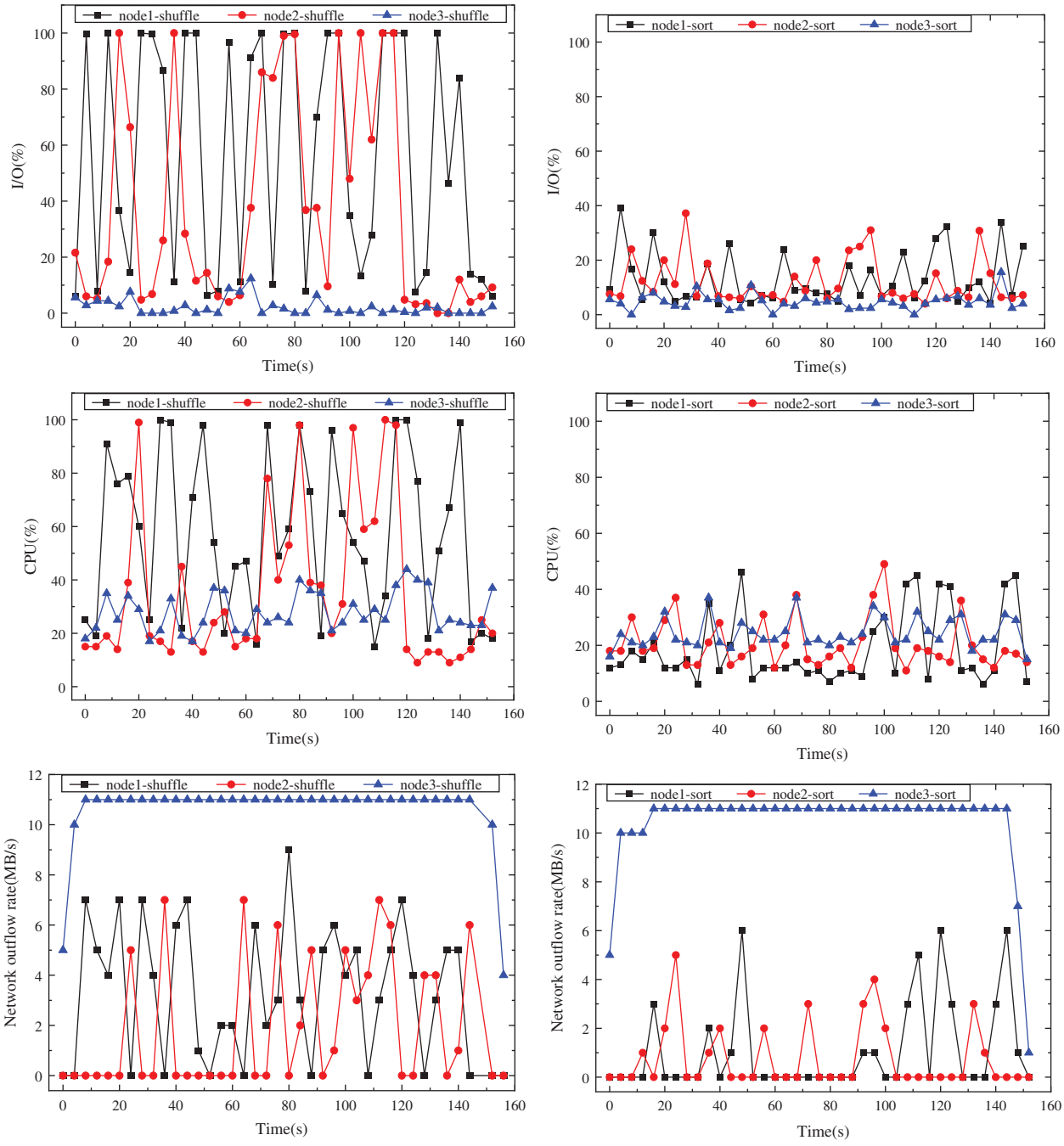

Figure 5: I/O conversion, CPU utilization, and network conversion rate at 30M/3s

From the above results, it can be found that the network outflow rate of storage node 3 was always very high. The reason is that storage node 3 had good machine performance and acted as a proxy node. All the data were returned to the user through the proxy node. Additionally, the network outflow rate of storage node

3 itself in the improved read strategy was higher than that that in Swift's own read strategy, indicating that proxy node 3 received more read requests because the improved read strategy allocated user requests according to the utilization status of storage I/O. The I/O performance of storage node 3 was better than that of storage node 1 and storage node 2, so more requests were allocated, also implying that the reading strategy proposed in this paper is also effective in scenarios with different storage node performances.

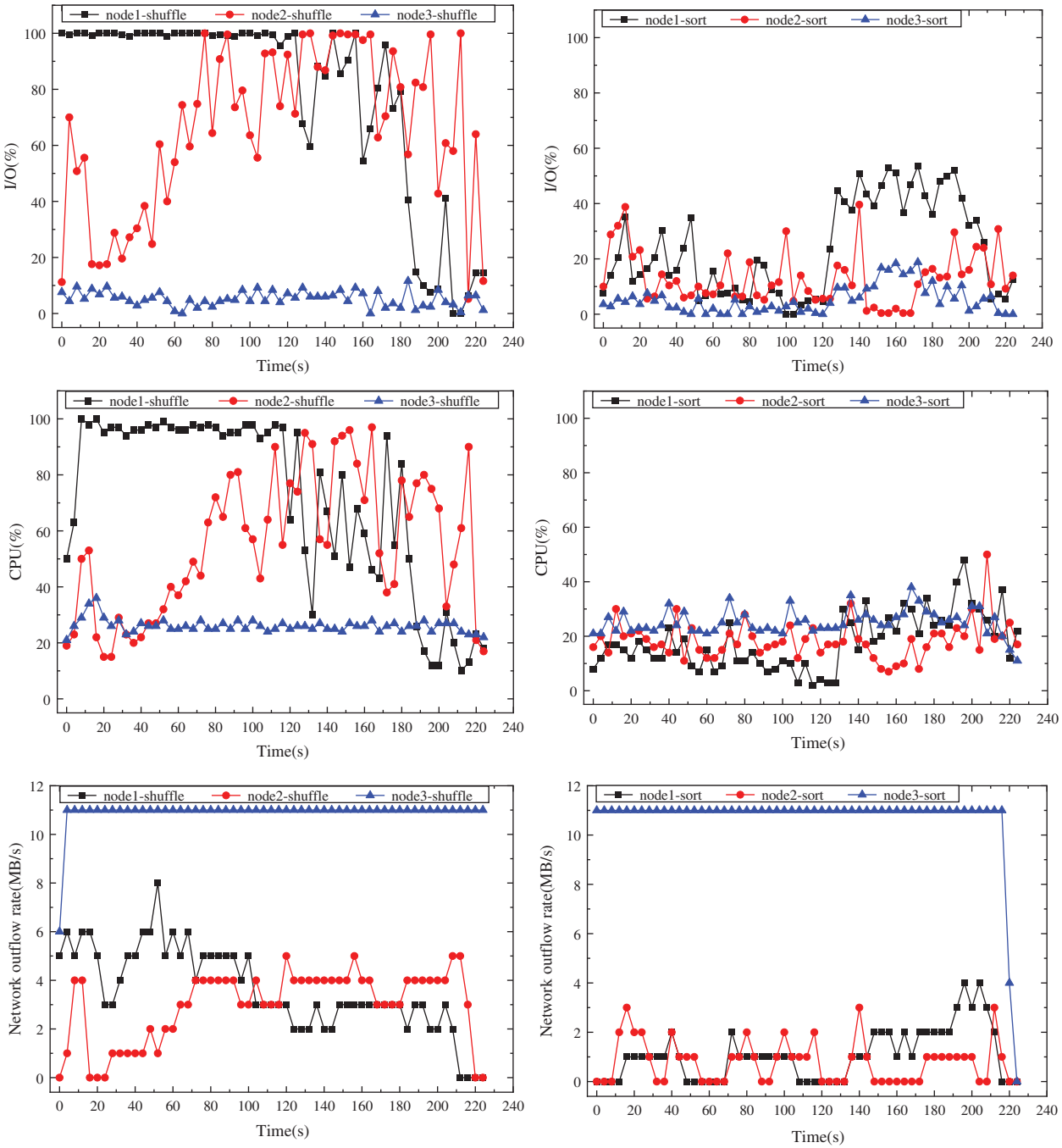

Figure 6: I/O utilization, CPU utilization, and network outflow rate at 50M/2s

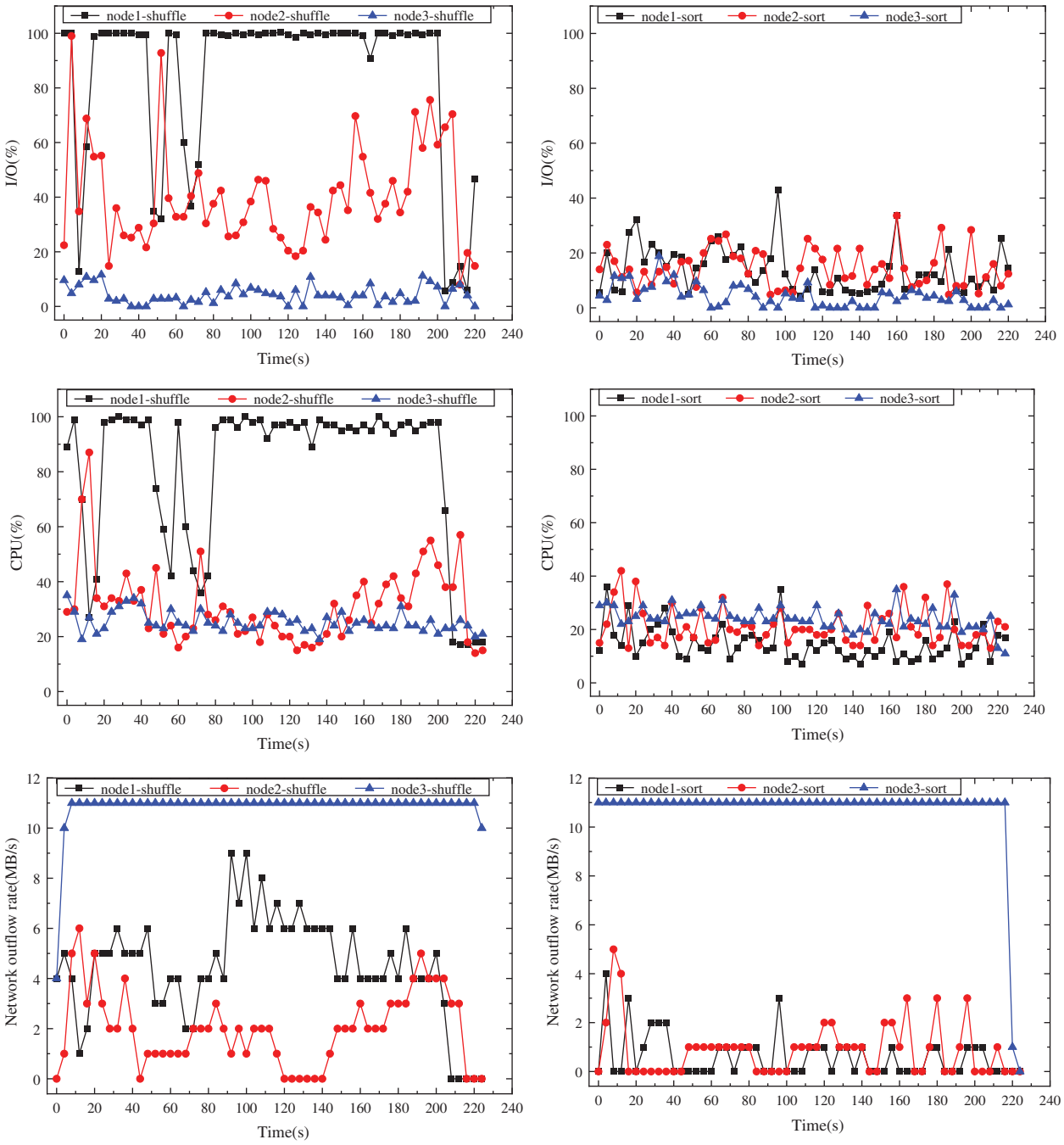

Figure 7: I/O utilization, CPU utilization, and network conversion rate at 50M/3s

It can be seen from Tab. 2 that the download time of the resource load balancing read strategy based on I/O utilization is less than the random read time of Swift, but the improvement is not significant. The reason is that the network speed during downloading has reached the bandwidth limit, which has become the performance bottleneck of the system, so the download time is not significantly improved.

Table 2: Download time in different scenarios

File size\time interval\read strategy	amount of users						
	5	10	15	20	30	40	50
30M\ 2s\ shuffle	19.85	37.82	44.90	57.30	83.81	114.24	142.29
30M \2s \sort	14.16	27.93	42.03	54.87	82.44	108.61	135.98
30M \3s\ shuffle	17.95	33.62	48.85	63.69	94.35	129.44	154.85
30M \3s \sort	16.81	31.65	45.94	62.00	91.19	122.49	153.97
50M \2s\ shuffle	23.06	49.22	68.10	92.62	141.17	184.34	232.74
50M \2s \sort	23.07	45.41	67.69	89.98	134.63	181.27	226.35
50M \3s \shuffle	24.86	47.51	73.72	94.83	141.12	183.81	232.90
50M \3s\ sort	24.57	45.63	68.33	92.28	135.31	180.24	226.22

5 Conclusion and Discussion

This paper mainly studied the reading strategy in Swift. First, by analyzing the implementation of data reading process, we found that Swift stored multiple copies and needed to specify the storage node to read the copy. The copy reading strategy adopted by Swift is a random choice without considering the resource load of storage nodes. To address this issue, this paper proposed a reading strategy based on I/O load information. According to the feedback on I/O utilization rate from the storage node, the storage node with the least I/O utilization rate is selected for providing the data copy requested by users, which solves the problem that some storage nodes are overloaded while other storage nodes are lightly loaded due to the random selection of data reading strategy in Swift. The experimental results show that, compared with the original data reading strategy in Swift, the strategy based on load balancing of storage node resources proposed in this paper can make the resource load of storage nodes more balanced.

The next step of the research can start from data migration such as monitoring the storage capacity of storage nodes to avoid excessive capacity of some storage nodes and low capacity of others and migrating the data to the storage nodes with larger capacity to balance their storage load.

Acknowledgement: The authors are grateful to the anonymous referees for having carefully read earlier versions of the manuscript. Their valuable suggestions substantially improved the quality of exposition, shape, and content of the article.

Funding Statement: This work is supported by the Fundamental Research Funds for the Central Universities (Grant No.HIT.NSRIF.201714), Weihai Science and Technology Development Program (2016DXGJMS15), Key Research and Development Program in Shandong Provincial (2017GGX90103) and Weihai Scientific Research and Innovation Fund (2020).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] D. J. Zhu, Y. D. Sun and H. W. Du, "HUNA: A method of hierarchical unsupervised network alignment for IoT," *IEEE Internet of Things Journal*, vol. 8, no. 5, pp. 3201–3210, 2021.

[2] Badshah and Afzal, "Smart security framework for educational institutions using internet of things (IoT)," *Computers Materials & Continua*, vol. 61, no. 1, pp. 81–101, 2019.

[3] S. Xiong, Q. Ni and L. Wang, "SEM-ACSIT: Secure and efficient multiauthority access control for IoT cloud storage," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 2914–2927, 2020.

[4] Q. Wang, F. Zhu and Y. Leng, "Ensuring readability of electronic records based on virtualization technology in cloud storage," *Journal on Internet of Things*, vol. 1, no. 1, pp. 33–39, 2019.

[5] D. J. Zhu, H. W. Du and Y. H. Wang, "An IoT-oriented real-time storage mechanism for massive small files based on Swift," *International Journal of Embedded Systems*, vol. 12, no. 1, pp. 72–80, 2020.

[6] P. Yang, N. Xiong and J. Ren, "Data security and privacy protection for cloud storage: A survey," *IEEE Access*, vol. 8, pp. 131723–131740, 2020.

[7] C. T. Yang, S. T. Chen and W. H. Cheng, "A heterogeneous cloud storage platform with uniform data distribution by software-defined storage technologies," *IEEE Access*, vol. 7, pp. 147672–147682, 2019.

[8] X. Shi, Y. Li and H. Xie, "An openflow-based load balancing strategy in SDN," *Computers, Materials & Continua*, vol. 62, no. 1, pp. 385–398, 2020.

[9] Y. Zhou and F. Liu, "Research on server load balancing technology," *Computer and Digital Engineering | Comput Digit Eng*, vol. 38, no. 4, pp. 11–14, 2010.

[10] R. Zhang, "Design and prototype implementation of load balancing scheme of DNS system," M.S. dissertation. Beijing University of Posts and Telecommunications, China, 2011.

[11] S. Lin, D. Luo and H. Zhang, "Research of load-balancing of web-server cluster based on genetic algorithms," *Computer Measurement and Control*, vol. 14, no. 10, pp. 1364–1365, 2006.

[12] Y. Yang, "Research on LVS load balancing scheduling based on adaptive niche genetic algorithm," M.S. dissertation. Southwest Jiaotong University, China, 2013.

[13] F. Li, "Research on cloud computing resource load balancing scheduling algorithm based on ant colony algorithm," M.S. dissertation. Yunnan University, China, 2013.

[14] C. Shi and Z. Li, "Research on load balancing algorithm based on improved polymorphic ant colony in Linux cluster," *Journal of Sichuan University: Natural Science Edition*, vol. 46, no. 5, pp. 1311–1315, 2009.

[15] L. C. Hu, Y. J. Xu and H. M. Xu, "Consistent hash load balancing algorithm based on dynamic feedback," *Electronics and Computer*, vol. 29, no. 1, pp. 177–180, 2012.

[16] Q. Wang, "Research on grid workflow scheduling algorithm based on load balancing," M.S. dissertation. Xiamen University, China, 2009.

[17] Q. Tan, "Research on load balancing strategy in cloud computing environment," M.S. dissertation. Xiamen University, China, 2014.

[18] L. Lv, "Research on resource load balancing scheduling algorithm in cloud computing environment," M.S. dissertation. Xinjiang University, China, 2010.

[19] H. Wang, "Research on adaptive load balancing scheduling strategy of web server cluster system," Ph.D. dissertation. Jilin University, China, 2013.