Research on Load Balancing Strategy of Standalone Multi-core Cluster based on Node.js

Bo Wu, Zhengping Jin

State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, 100876, China

coolwubo@163.com, zhpjin@bupt.edu.cn

Keywords: Node.js, Load Balance, Request Classification, Weighted Round Robin.

Abstract. With the development of Internet and the rapid increase of users, there are a lot of serious problems in Internet, such as network congestion, server overload and too long response time. So it is necessary for network applications to support the high concurrent access requirements. Node.js is an asynchronous and non-blocking I/O model that provides efficient scheme for this problem. However, as for the Node.js native single-threaded work pattern cannot use performance of multicore CPUs, the advantage of high concurrency cannot given full paly to. In this paper, we proposed a simple and efficient Node.js multi-core and parallel framework based on the study of the existing multi-core solutions. This framework balances the nodes through the way of request classification and weighted round-robin. The strategy consider the difference between the user requests and the working process of real-time load totally. Moreover, at the same time we introduce the load feedback mechanism, which can balance the load of nodes and make full use of the resources of each node better through allocating user request according to equivalent transformation of the load weight and accelerated decline weights of critical state. Experiment show that the plan has better load balance ability compared to the the original solutions under the condition of high concurrent requests.

Introduction

To settle the current bottleneck problems in the performance of high-concurrency Web application system, the solution involves in the major aspects as follows: improvement of the performance and quantity of server, promotion of database performance, application of database cluster, further improvement of vessel performance, cache optimization and use of message queue, etc. However, those fail to improve the performance of HTTP server greatly.

As an emerging and hot server-side technology, Node.js technology adopts the event-driven based asynchronous I/O model, which greatly improves the concurrency performance of HTTP server [1]. However, it has some disadvantages, e.g., failure of its single-process model in adequate using the advantages of multi-core server and low reliability [2]. For the purpose to solve the problem, Cluster module is introduced in Node.js V0.8.0, and it adopts the operating system for access request scheduling and load balancing [3]. Experimental results show that the method is still imperfect in load balancing, easy to result in uneven request dispatching and has a certain limitation under the complex network environment [4]. Therefore, the emphasis was laid on solution to multi-core CPU resource utilization of Node.js so that it can get the most out of CPU resource and achieve the excellent load balancing with a large amount of concurrent access request.

In this paper, a simple and effective solution was proposed, on the basis of common load balancing strategies and Node.js features, for balancing the load on various nodes, by means of request classification and weighted round robin. In the strategy, adequate consideration was given to the differences in user requests and real-time loads between various worker processes. The controller is introduced to identify the types of client requests and then the same kind of requests will be allocated to the servers on various nodes in a uniform way, so that the quantity of various requests received by each node server is roughly the same. Moreover, the feedback link was introduced to allocate the user requests rationally by means of periodic equivalent transformation of load weights and accelerating weight decreasing after entry into the critical state, to balance the node load and make full use of various node resources. Experimental results show that the improvement plan proposed in this paper,

when compared with Cluster module, has a better load balancing effect under high-concurrency request.

Analysis of Solution to Load Balancing by Node.js Cluster Module

Node.js V0.8.0 has built-in third-party Cluster module for realizing Node multi-core processing and distribution of the user's request. Cluster realizes Node multi-core HTTP server through the shared ports, i.e., a socket shared and a port is monitored by multiple processes. The solution is realized depending mainly on the operating system kernel. The architecture is shown in Fig. 1[5].

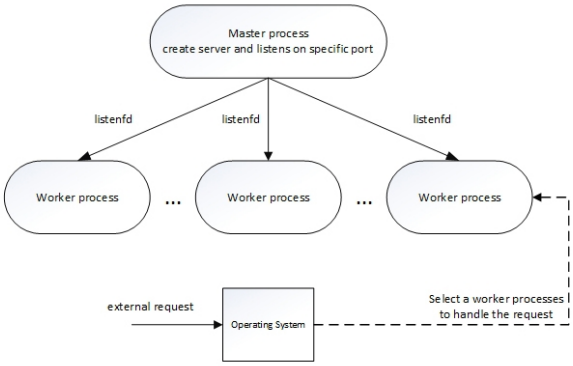

Fig. 1 Node multi-core server architecture base on cluster

Fig. 1, the main process master creates a socket for monitoring the specific ports and sending its corresponding file descriptor (handle) to all subprocesses workers. After receiving the handle, each worker will incorporate it into the respective Epoll monitoring structure, so that multiple worker processes share one port. When an external request comes, the operating system decides which worker process will obtain the activation event and receive and handle the request. These workers are mutually independent processes and are subject to direct scheduling and management by the operating system, and each worker runs on the separate CPU core to realize the Node multi-core server and make full use of the computing power of multi-core CPU [6].

No intermediate layer is introduced in the architecture, so the Node multi-core framework is established in an extremely concise way; In addition, the process, in which the operating system allocates which worker process will obtain the activation event, is highly efficient. However, the unbalanced loading exists in the practical application of Cluster.

The load balancing in Cluster depends completely on the scheduler of operating system. All workers monitor the same socket and after a new request comes the operating system will allocate it to a worker. The operating system will collect various data and indicators for process running, so it is best qualified to schedule the worker process in theory and can ensure the fair request distribution between various workers. However, it seems that the complaint in unbalanced Cluster load never stops. According to the test for Cluster load balancing effect, most requests will be allocated to several specific workers. The realization of operating system scheduler is quite complex, so only the following possible reasons are analyzed [7]: For the operating system, it tries to avoid the expensive context switch as far as possible. If n workers are waiting for the same socket, the operating system will wake up the recently blocked worker, to minimize the opportunity of context switch.

In conclusion, the reasons for the problem is: the operating system and programmer have different views of the problem, and the operating system thinks that the best choice will not certainly meet the programmer’s demands, which results in the Cluster unbalanced loading. The load balancing is quite important for a high-concurrency cluster.

Request Classification and Weighted Round Robin-based Solution

In the solution proposed in this paper, Node is applied for main process realization, i.e., distributor process (dispatcher) and N subprocesses, namely, worker process (worker). The dispatcher process is intended for monitoring the port; receiving the external request and distributing the well-connected

socket file descriptor to the worker process which will process the request prior to returning the processing result to the client. The dispatcher is intended for forwarding the external request to worker process in a balanced way and for external request management.

For the purpose to promote the effect of load balancing, the request classification and weighted round robin were adopted in this paper for balancing worker load. In the system, the dispatcher functions as a scheduler to collect all central scheduler: Firstly, the requests will be classified according to request URL and other application layer information and then distributed to the different queues of workers according to the different CPU intensive requests and I/O intensive requests. The weighted round robin (WRR) is adopted during request distribution, so that the quantity of various requests received by each node server is approximately the same as its weight. To avoid the unbalanced node load as far as possible, the controller will monitor the actual loads of various node servers periodically, equivalently calculate the current load weights of various node servers by means of acquisition of response time of various node servers and then adjust the distribution ratio based on the actual load weights. Each worker has its own scheduling queue and thus it will take a request from the scheduling queue for implementation after the current task is completed. These scheduling queues are applied for the communication between the dispatcher and various workers.

Request classification. WEB site has many static pages as well as a lot of tasks adopting dynamic object embedded technology and database operation, so different request tasks possibly consume different system resources. To achieve the load balancing, the user requests shall be classified. Featured by non-blocking asynchronous I/O, Node.js is efficient in processing I/O intensive requests but is inefficient in processing CPU intensive tasks, so we classify the Web services into two types as follows:

1) I/O intensive: Provide the static information (e.g., HTML page and embedded object) and request for result from dynamic database query, and generally the query condition is provided in a dynamic way by the user through HTML page. The intensive disk access is required, so it is called also as I/O intensive service.

2) CPU intensive: Provide the dynamic information and secure information transmission, focusing in e-commerce application. For security reasons, some dynamically generated data shall be transmitted via secure paths, and in most cases SSL protocol is used. The encryption and decryption operations consume a lot of CPU resources, so the kind of service is regarded as the CPU intensive service.

Weight calculation. For the traditional load balancing strategy, the acquisition of performance parameters, e.g., CPU, memory, system I/O rate and network bandwidth of various server nodes, is required for the integrated computation of the weights of various nodes[8]. There is no difference between various workers of Node.js standalone cluster, so a relatively simple and effective method is proposed for equivalent transformation of weight, to reduce the system resource consumption and improve the cluster efficiency.

Analysis based on a single worker's response status and response time:

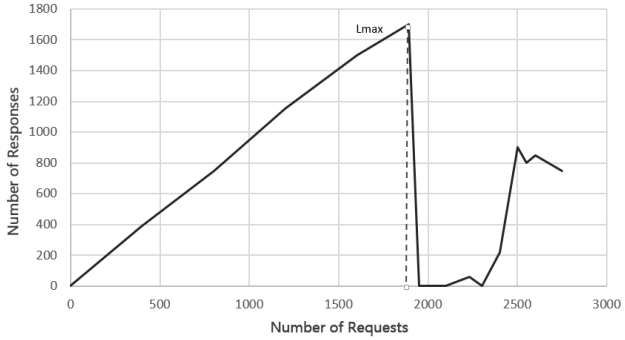

Fig. 2 Relational graph of the number of request sent and the number of request receiving response

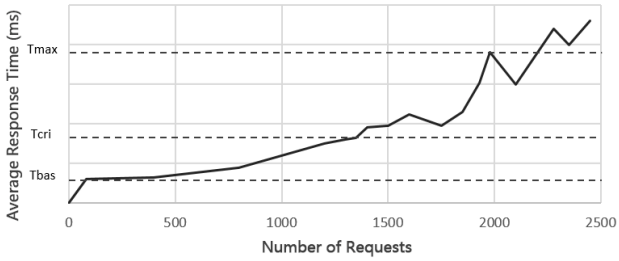

Fig. 3 Relation between the number of request sent and the average response time

It can be seen from Fig. 2 and Fig. 3 that if the number of request sent reaches to a certain value Lmax, the number of request receiving server response will drop to zero and keeps for a period. Which is a kind of measure taken by the system software to prevent the dead halt and the phenomenon is called as "false dead halt", and it will cause the severe impact on the server performance and therefore shall be avoided as far as possible. In case of "false dead halt" of serve, the response time will sharply increase and exceed Tmax, and under the condition, the server's available resource can be regarded as zero. Fig. 3 shows that when the most basic request task is loaded to the no-load server, server's response time is Tbas, and if the request impact is neglected, Tbas can be regarded as the basic load for the server to keep running. We might as well define (Tmax-Tbas) as the external service capacity can be provided by server and set Tnow as the current response time of the server, and then (Tmax-Tnow) represents the current left system capability of service of the server. Therefore, the equivalent transformation of current weight of node *i* is shown as follows, where: Load (N*i*) is the initial weight of the corresponding node, and the initial weight of various nodes in Node.js is the same:

$$W_i^{'} = Load_i \times \frac{T_{max}^i - T_{now}^i}{T_{max}^i - T_{bas}^i} \quad (T_{max}^i \geq T_{now}^i \geq T_{bas}^i) \tag{1}$$

Fig. 3 shows that if the number of request sent reaches to a certain value, the change of corresponding response time will jitter, indicating that the system will enter into the saturation condition. And then the zone can be defined as the critical zone prior to entry of the system into saturation condition, as shown in Tcri - Tmax from response time perspective. To restrict the server from entry into saturation condition, the workload at the current phase may be reduced by appropriate reduction in weight of node entering into the node into the critical zone. The adjustment formula is shown as follows:

$$W_i = W_i^{'} \times K, K = \begin{cases} \alpha\left(1 - \frac{T_{now}^i}{T_{max}^i}\right), T_{now}^i \geq T_{cri}^i \\ 1, T_{now}^i < T_{cri}^i \end{cases} \tag{2}$$

K is the adjustment coefficient and will act only after entry of node into the critical zone. The heavier node load, the smaller K value, for purposes to accelerate the decrease speed of node with large critical depth and to avoid the entry into the saturation condition. α is regarded as a constant for adjustment of decrease speed.

Request distribution. The dispatcher may obtain the current weights of various nodes according to their response time and then performs scheduling on the user request in various queues by means of weighted round robin(WRR), i.e., distributing the user requests to various nodes by means of round robin and according to the node weight degrees. The node with a high weight obtain more user requests and the quantity of user request allocated to each node is determined by the ratio of its weight to the sum of node weights.

In addition to distributing the external request to various workers in a "uniform" way, the dispatcher also conducts the efficient management of it. The main process of dispatcher shall master

the information relevant to worker process, i.e., survival status. The specific method adopted in this paper: all workers will feedback the heartbeat to dispatcher regularly and collect their respective information and report it to dispatcher. All workers shall feedback the following information: worker status and CPU of intensive requests, etc. The architecture is shown in Fig. 4.

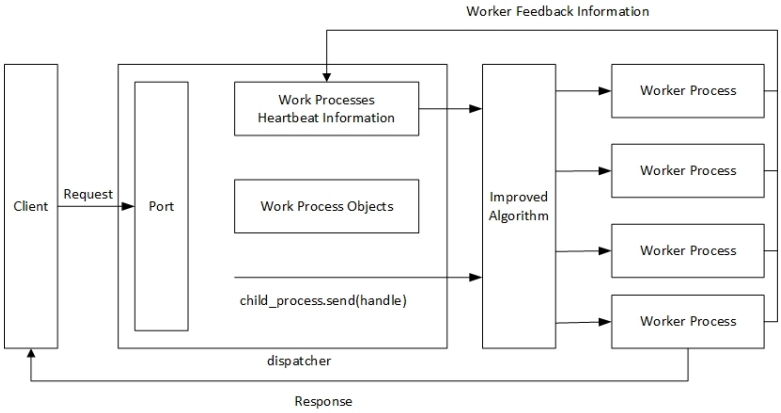

Fig. 4 Node multi-core server architecture

Experiment and Analysis

To verify the improvement effect of the load balancing of Node.js's Cluster module, the web application stress tool Jmeter, developed by Apache, is applied in the experiment for simulating the users' massive concurrent access. For the purpose to simulate the different types of method requests in network, we simulate a CPU intensive task by calculating Fibonacci sequence and simulate the I/O intensive task by file reading. We compare the load balancing effect based on Node.js's Cluster module with that based on our "Minimum Number of Requests" solution. There are two major evaluation indexes for measuring the load-balancing algorithm: average throughout and average response time. Performance contrast test result is shown in the Fig. below:

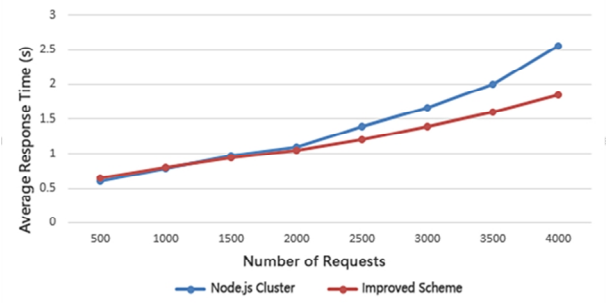

Fig. 5 Comparison of Average Time

Fig. 6 Comparison of Average Throughput

The contrast test results are shown in Fig. 5 and Fig. 6. The results show that if the load is light, the algorithm performance after improvement is basically similar to or slightly lower than that of cluster module, because that the algorithm after improvement requires the parsing and classification of the user request prior to forward, which results in a large system overhead. However, the cluster module

needs no understanding the specific contents of user request and may conduct the forwarding directly, so it has a small system overhead. As the load increases, the algorithm after improvement becomes superior to cluster with regard to performance.

Conclusions

Load balancing strategy is the key to improve the overall system performance. In this paper, a simple and effective solution was proposed, on the basis of common load balancing strategies and Node.js features, for balancing the load on various nodes, by means of request classification and weighted round robin. The proposed solution has the following features:

(1) The proposed algorithm fully considered the differences between the user requests and the working process of real-time load, through the controller to identify the type of client request, then the similar request evenly assigned to each node server, so that each node server to get all kinds of requests take roughly the same.

(2) Moreover, the feedback link was introduced to allocate the user requests rationally by means of periodic equivalent transformation of load weights and accelerating weight decreasing after entry into the critical state, to balance the node load and make full use of various node resources.

Two kinds of scheduling algorithm under different task load average throughput and the average response time is shown in Fig. 5 and Fig. 6. It can be seen that the improved pla by accurately measure the real-time load condition, reasonable allocation of live load, has better performance.

Acknowledgements

This work is supported by NSFC (Grant Nos. 61300181, 61502044), the Fundamental Research Funds for the Central Universities (Grant No. 2015RC23).

References

[1]. Tilkov S, Vinoski S. Node.js: Using JavaScript to Build High-Performance Network Programs[J]. Internet Computing IEEE, 2010, 14(6):80-83.

[2]. Ling Pu. Head First Node.js [M]. Beijing : Posts & Telecom Press,2014. In Chinese.

[3]. Information on https://nodejs.org/documentation

[4]. Paudyal U. Scalable web application using node. js and couchdb[J]. 2011.

[5]. M. Cantelon, and TJ. Holowaychuk, “Node.js in action”, Manning, 2011

[6]. M. Lehmann. “Libev”. http://software.schmorp.de/pkg/libev.html, 2012,

[7]. Pearce O, Gamblin T, de Supinski B R, et al. Quantifying the effectiveness of load balance algorithms[C]//Proceedings of the 26th ACM international conference on Supercomputing. ACM, 2012: 185-194.

[8]. Mathew V, Sitaraman R K, Shenoy P. Energy-aware load balancing in content delivery networks[C]//INFOCOM, 2012 Proceedings IEEE. IEEE, 2012: 954-962.