A modified ingress controller for Kubernetes

Jianxin Zhang[1]
{zjx556}@sjtu.edu.cn
[1]Shanghai Jiao Tong University

Abstract

Recently web applications have grown explosively with the widespread of internet and networking devices. Many services providers rely on container-base clusters like kubernetes to orchestrate their web services. The container-base clusters bring convenience to applications lifecycle management, but it is disappointing that they do not combine their advantages to acheive a better load balancing performance. This paper observes that for a specific application with diverse resource consumption (such as CPU, memory, etc.) and frequency of different kinds of requests, it is of high possibility that load imbalance between backend servers for different system resources may happen, so problems like performance degradation will be caused ineluctably if too many requests are sent to over-load servers. To address these problems, this paper proposes a dynamic load balance solution based on kubernetes [11] which takes request consumption and server load into account is proposed to reduce the performance problem. Experimental results are given to validate the performance of proposed solution.

1 Introduction

The rapid growth of internet introduces a new economy pattern named internet economy, under which almost all activities can be supported by web services through PCs or mobile devices. Aiming to achieve high resource utilization and better lifecycle management, web applications tend to be deployed into cluster based on a lightweight virtualization called container just like other traditional applications do. Unlike traditional applications which only execute job assigned in advance, web services have to listen on users' requests and responses as soon as possible. Workload of web services are changing unpredictably because they offer varied businesses, each business consists of operations that may consume CPU, allocate memory or access disk, and it is impossible to know which business will be executed in any time. The unpredictable workload introduces a hug challenge on load-balancing. Unfortunately, the container-based solutions like Kubernetes fail to cope with this challenge, their fair load-balancing algorithm performs well only when workload are the same, which is contrary to workload of web services, as a result, web services fail to gain high resource utilization, it loses stability and performance conversely.

To address the load balance problem, this paper introduce a dynamic load-balancing solution based on Kubernetes, a popular container cluster management system. The new solution mainly covers two issues: Load collection and load-balancing algorithm.

The load collection part tries to collect information for load balancer to adjust its requests dispatching decision.For this part, we implement a load monitor to evaluate resource consumption of request types and collect load of backend containers. Through collecting and analyzing logs of web services, the monitor generates resource description for different request types, which represents their consumption of different resources. Through listening on status changes of container, monitor generates load description for usage of CPU, memory and network.

For the algorithm part, we introduce request and container models based on description generated by monitor. The generation of models considers not only the load description, but also other details such as the locality, deployment configuration of container and so on. Request dispatching is done through matching request models to container models.

In summary, this paper makes the following contributions:

- A load monitor for learning resource consumption of requests and listening load changes of containers.
- A dynamic load-balancing algorithm which com-

bines resource consumption of request and real-time load of container to dispatch request and finally achieve stability, high performance and high resource utilization for web services.

- An evaluation to show the performance of dynamic load-balancing solution.

The rest of the paper is organized as follows: Section 2 describe background information about performance of web services in container-base cluster. Section 3 presents the design and implementation of dynamic load-balancing solution. Then section 4 evaluate the solution. Section 5 shows the related work, and finally section 6 discuss on the future work.

2 Background

Performance of web services is sensitive to system metrics of their backend servers, in container-based cluster, that is backend containers. For convenience, we called the backend containers as endpoints in this paper. This section first points out factors that affect performance of web services in container-based cluster, and then introduces the current design of load-balancing solution, that is ingress controller in Kubernetes and finally state defects of ingress controller.

2.1 Influencing factors

Factors that affect the performance include CPU usage, memory usage, network usage, services dependency and locality. As known to us, servers with high usage in CPU, memory or network will perform badly. Things are same when it comes to container-based cluster, for an endpoint, it causes delay on requests processing when it is high-loaded in CPU and network. More and more, if the endpoints are in high-loaded state for memory, they face the risk that they may be restarted due to OOM(out of memory) problem. Thus, if we demand high performance for web application, we'd better even load of each endpoint and reduce overhead endpoints as many as possible. services dependency means that services do not provides functions themselves, they rely on applications of other services to finish their jobs. For example, we have endpoint1, endpoint2,endpoint3 for serviceA, and endpoint4,enpoint5 for serviceB, and serviceA relies on serviceB to process its requests, particularly, endpoint1 and endpoint2 rely on endpoint5, endpoint3 relies on endpoint5. Obviously, the performance of endpoint4 and endpoint5 will directly influence the performance of endpoint1, endpoint2 and endpoint3. Furthermore, the competition between endpoint1 and endpoint2 for accessing endpoint5 will have a negatice impact on performance of serviceA. This paper select database dependency as an example of services dependency. That is endpoints of a web service rely on endpoints of database to process requests, which is also a common case for web services. Traditionally, developers tend to reduce pressure of database by dividing it into several databases and synchronize data between them. Therefore, it is common for an endpoint to access a database that is not only the same as some endpoints, but also different from others. Under this circumstance, performance of web applications can be improved through reducing race between endpoints that share the same database or reducing the overhead endpoints of databases. Locality means that the placement of endpoints may affect their performance. Though containers are claimed to isolate resource from each other, papers [14] still show that when two or more endpoints are placed in the same node, the delay appears. More and more, although under circumstance that an endpoint monopolizes a node, it still remains possibility for delay when the node is in high-load status because of some system processes. So the locality should also be considered for a better load-balancing performance.

2.2 Ingress controller

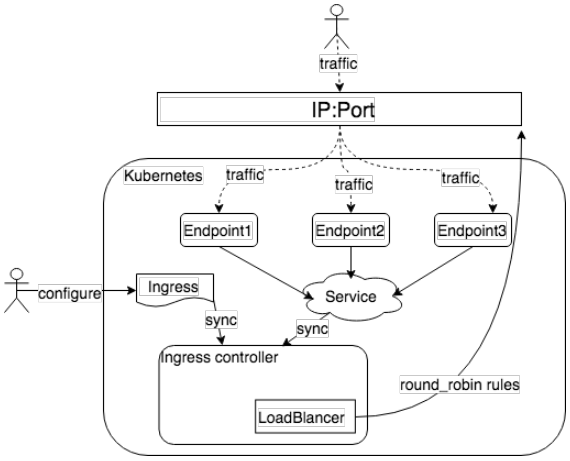

Figure 1: Workflow of ingress controller

Figure 1 shows the workflow of kubernetes load-balancing component – ingress controller: To balance network workload, ingress controller focuses on two part, one is to keep track of the IP and number change of endpoints so that it can update traffic rules in time to guarantee accuracy. The other part is watching ingress to update load-balancing strategy. Ingress is a resource describing inbound rules for connections to reach endpoints. With correct status of endpoints and a configured ingress, ingress controller sets up appropriate traffic rules

for its built-in loadbalancer(nginx). Finally, the loadbalancer dispatches requests to suitable endpoints according to configured rules, balancing load across endpoints.

2.3 Shortcomings of ingress controller

Rather than balancing load, ingress controller concentrates more on linking ingress to nginx features like session affinity to let users to custom their load-balancing algorithm smoothly in kubernetes. However, its configurable algorithms are simply based on static algorithms like round_robin, which do not consider influencing factors mentioned above. To summary, the igress controller has two shortcomings when it is put into pratice, that is hysteresis and one-sidedness. The round_robin strategy assumes that all requests have the same resource consumption and aims to balance load after a long period of time, so problems come when requests change in a rather short time. For example, when there is a peak of requests, and new endpoints is needed to participate in providing service. A appropriate way to balance load now is to dispatch more workload to new idle endpoints and reduce workload for high-loaded old endpoints, but the round_robin algorithm still dispatches requests indistinguishably, taking the riks that the old high-loaded endpoints may be unable to work normally, even restarted due to OOM although it will finally balance load relatively. The one-sideness means that the round_robin algorithm only considers to even workload accross endpoints, without taking influencing factors mentioned above, thus failes to balance load accurately. To address these shortcomings of ingress controller, this paper presents an alternative basic algorithm to ingress controller, that is a dynamic load algorithm that takes these factors and request information into consideration.

3 Design and implementation

To overcome the shortcoming of ingress controller, this paper proposes a dynamic load-balancing solution. The solution consists of two part, one part is load monitor, the other part is dynamic load-balancing algorithm.

3.1 Workflow

Figure 2 illustrates the workflow of the dynamic load-balancing solution:

(1) Load monitor gather necessary data for dynamic load balancing and offer data to loadbalancer.

(2) Loadbalancer makes a further evaluation for data received from loadmonitor and dispatches incoming requests based on dynamic load-balancing algorithm.

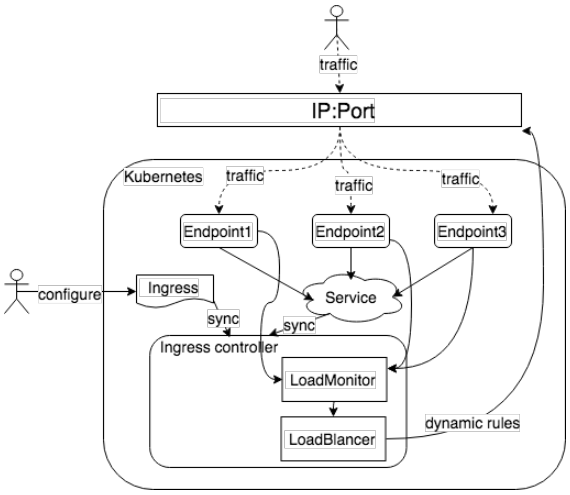

Figure 2: Workflow of dynamic solution

3.2 Load monitor

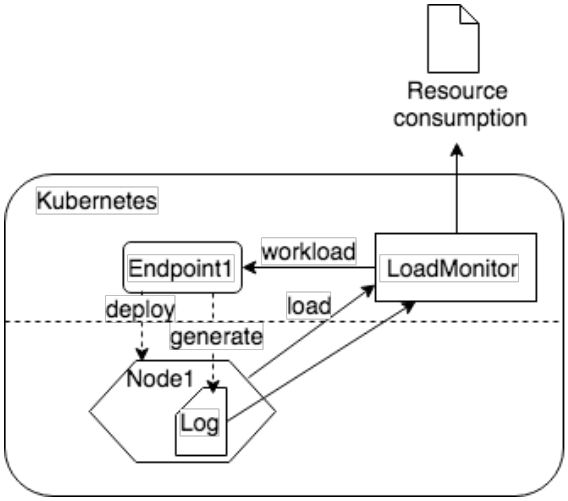

Figure 3: request-learing phase

Load monitor collects, analyzes and outputs information needed by dynamic load-balancing algorithm. Load monitor processes two kinds of information: request logs and endpoint loads. Request logs are used for evaluating resource consumption of request. Before running in kubernetes, each web service goes through the request-learning phase shown by figure 3: First, a single endpoint is deployed, then load monitor generates workload by keep on sending requests in a fixed speed to the endpoint, the speed increases with time until it reach the processing limit of endpoint. At the same time, load monitor combines service logs, workload and endpoint load to evaluate resource description for each type of request. We define RD_i as resource description of a i_{st} type request.C_i ,M_i and N_i stand for CPU, memory, network consump-

tion to process a i_{st} type request. S_i^t stands for speed of sending i_{st} type request in timestamp t. c_i^t, m_i^t, n_i^t are CPU, memory, network load of endpoint in timestamp t during i_{st}request-learning phase. The computation of resource description is as follow:

$$C_i = \frac{\sum_{j=1}^{k} \frac{c_i^t}{s_i^t}}{k}, M_i = \frac{\sum_{j=1}^{k} \frac{m_i^t}{s_i^t}}{k}, N_i = \frac{\sum_{j=1}^{k} \frac{n_i^t}{s_i^t}}{k}$$

$$RD_i = \{C_i, M_i, N_i\}$$

Then load monitor enters load-collecting phase shown in figure 4 to generate load description for endpoints.To increase load-balancing performance, load description of endpoints has to cover all influencing factors. For the resource usage part, Load monitor collects system loads from kubelet, a component that is not only responsible for managing lifecycle of pods, but also takes charge of collecting cluster load. For sharing problem and locality, load monitor obtains endpoint information from apiserver. Endpoint information includes configuration that which endpoints share the same database, which endpoints are deployed in the same node and so on. Table 1 is an example of load description given by load monitor.

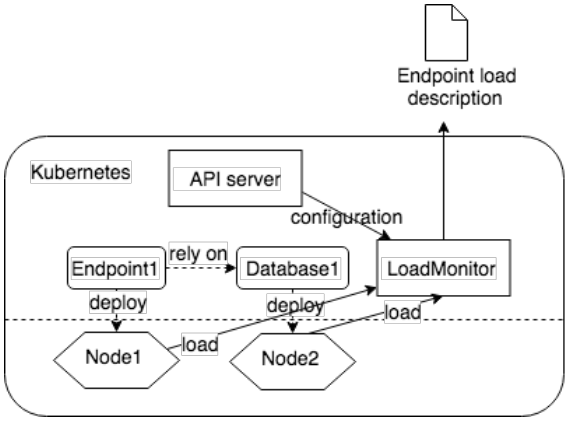

Figure 4: load-collecting phase

3.3 Dynamic load-balancing algorithm

The dynamic load-balancing algorithm considers request consumption and endpoint load to even workload across endpoints. It aims to reduce overload endpoints and promote RPS. The algorithm is divided into two phase: Classification phase and dispatching phase. During classification phase, we evaluate requirement of requests, utilization of nodes and ability of endpoints and classify them. For evaluating requests, we let requests compare

Type	Pod	Pod	Node
Name	App1	Db1	Node1
CPU limit	10	10	50
CPU usage	2	3	20
Memory limit	100M	200M	2G
Memory usage	20M	100M	500M
Network limit	1M/s	2M/s	5M/s
Network usage	500k/s	1M/s	3M/s
Node	Node1	Node1	-
Database limit	Db1	-	-

Table 1: Load description

with each other to get their consumption level of CPU, memory and network, that is to know whether their consumption is high or not compared to others. For evaluating node, we simply adopt utilization rate under the assumption that hardware configuration of all nodes in kubernetes are the same. For the pod part, first we compare usage of CPU, memory and network among pods to get their load level, then we take load level, utilization rate and ability of node and database that pods rely on in to consideration to evaluate ability of endpoints. The evaluation is as follow:

1. Calculate requirement level of each request type in CPU, memory and network:

$$L_{max} = Max\{L\_f_1, ..., L\_f_i\}$$

$$L_{min} = Min\{L\_f_1, ..., L\_f_i\}$$

$$r\_f_i = \frac{L\_f_i - L_{min}}{L_{max} - L_{min}} \cdot 100\%$$

$$R\_f_i = \begin{cases} High & r\_f_i \geq T_r \\ low & r\_f_i < T_r \end{cases}$$

where L_f_i is consumption of a given factor among CPU, memory and network, r_f_i represents consumption of i_{st} request type among all request types.R_f_i is requirement level. For each request type, its requirement is $\{R\_CPU, R\_MEM, R\_NET\}$. By default, we set the threshold $T_r = 0.5$.

2. Calculate utilization of node in CPU, memory and network:

$$U_i = \frac{L\_usage_i}{L\_limit_i} \cdot 100\%$$

where L_usage_i and L_limit_i denote usage and limit of a given factor among CPU, memory and network, U_i represents utilization of i_{st} node.For each node, its utilization is $\{U\_CPU, U\_MEM, U\_NET\}$.

3. Calculate ability of pod in CPU, memory and network:

$$L_{max} = Max\{L\_usage_1, ..., L\_usage_i\}$$

$$L_{min} = Min\{L\_usage_1, ..., L\_usage_i\}$$

$$U_i = \frac{L\_usage_i}{L\_limit_i} \cdot 100\%$$

$$P_i = \frac{L\_usage_i - L_{min}}{L_{max} - L_{min}} \cdot 100\%$$

$$A_i = 1 - (U_i \cdot \alpha + P_i \cdot \beta + N_i \cdot \gamma + D_i \cdot \delta)$$

$$(\alpha + \beta + \gamma + \delta = 1)$$

where L_usage_i denotes usage of a given factor, L_limit_i is the limit, U_i stands for utilization rate, P_i represents load level, N_i and D_i are utilization of node and ability of database that pod relies on. For database pods, the D_i is zero.By default, we set $\alpha = 0.5, \beta = 0.1, \gamma = 0.2, \delta = 0.2$. For each pod, its ability is $\{A\_CPU, A\_MEM, A\_NET\}$.

During dispatching phase, endpoints are sequenced by their ability: Firstly, we set up thresholds to group endpoints. That is $T_c = 70\%$ for deciding whether endpoint's CPU ability is in high or normal level, $T_m = 50\%$ for deciding whether endpoint's memory ability is high or normal, $T_n = 50\%$ for deciding whether endpoint's network ability is high or normal. As a result, endpoints are grouped into 8 groups. Table 2 shows definition of these groups.

Group	CPU	Memory	Network
g1	$A\_CPU \geq T_c$	$A\_MEM \geq T_m$	$A\_NET \geq T_n$
g2	$A\_CPU \geq T_c$	$A\_MEM \geq T_m$	$A\_NET < T_n$
g3	$A\_CPU \geq T_c$	$A\_MEM < T_m$	$A\_NET \geq T_n$
g4	$A\_CPU \geq T_c$	$A\_MEM < T_m$	$A\_NET < T_n$
g5	$A\_CPU < T_c$	$A\_MEM \geq T_m$	$A\_NET \geq T_n$
g6	$A\_CPU < T_c$	$A\_MEM \geq T_m$	$A\_NET < T_n$
g7	$A\_CPU < T_c$	$A\_MEM < T_m$	$A\_NET \geq T_n$
g8	$A\_CPU < T_c$	$A\_MEM < T_m$	$A\_NET < T_n$

Table 2: Endpoint groups

A new endpoint is inserted into a desired group by insertion sort algorithm in descending order first by A_CPU, then A_MEM and finally A_NET. The algorithm 1 describes the insertion.

The dispatch decision is made based on request requirement level and endpoint groups. The steps of dispatching a request is as follow:

Algorithm 1 modified insertion sort algorithm

Require: $g[]$, $endpoint$
1: $i \leftarrow 0$
2: **while** $i < sizeof(g)$ **do**
3: **if** $!(endpoint.A\_CPU \geq g[i].A\_CPU)$**and**
4: $(endpoint.A\_MEM \geq g[i].A\_MEM)$ **then**
5: $i++$
6: continue
7: **else if** $endpoint.A\_CPU \geq g[i].A\_CPU$ **then**
8: break
9: **end if**
10: i++
11: **end while**
12: DO_INSERTION($g, i, endpoint$)

Algorithm 2 selection algorithm

Require: $hg[\{ep | ep\ \varepsilon\ g1 \cup g2 \cup g3 \cup g5\}]$,
$lg[\{ep | ep\ \varepsilon\ g4 \cup g6 \cup g7 \cup g8\}]$,$request$
1: **function** RANDOM_CHOICE(hg,lg)
2: **if** $sizeof(hg) == 0$ **then**
3: **if** $sizeof(lg) \leq 3$ **then**
4: **return** lg
5: **else**
6: $i, j, k =$ RANDOM_3INT$(0, sizeof(lg))$
7: **return** $\{lg[i], lg[j], lg[k]\}$
8: **end if**
9: **else if** $sizeof(hg) \leq 3$ **then**
10: **return** hg
11: **else**
12: $i, j, k =$ RANDOM_3INT$(0, sizeof(hg))$
13: **return** $\{hg[i], hg[j], hg[k]\}$
14: **end if**
15: **end function**
16: **function** SCORECANDIDATE(candidate,request)
17: $i \leftarrow 0$
18: **while** $i < sizeof(candidate)$ **do**
19: $S\_c \leftarrow candidate(i).A\_CPU * request.\theta$
20: $S\_m \leftarrow candidate(i).A\_MEM * request.\lambda$
21: $S\_n \leftarrow candidate(i).A\_NET * request.\mu$
22: $candidate(i).Score = S\_c + S\_m + S\_n$
23: **end while**
24: SORTBYSCORE$(candidate)$
25: **return** $candidate$
26: **end function**
27: $c \leftarrow$ RANDOM_CHOICE(hg, lg)
28: $c \leftarrow$ SCORECANDIDATE(c)
29: **if** $request.priority == High$ **then**
30: DISPATCH_REQUEST$(request, highestof(c))$
31: **else**
32: DISPATCH_REQUEST$(request, middleof(c))$
33: **end if**

1. Query request type and corresponding requirement level.

2. Select suitable endpoint for request, the selection is decided by selection algorithm 2. First of all, we define priority of endpoint groups: we regard g4,g6,g7,g8 as low-priority groups and g1,g2,g3,g5 as high-priority groups because of the fact that endpoints have poor performance when they score badly in more than one factor. Then we pick three endpoints as candidates randomly from high-priority groups, grade them and choose the most suitable one. The equation to grade endpoints is as follow:

$$G_i = A\_CPU_i \cdot \theta + A\_MEM_i \cdot \lambda + A\_NET \cdot \mu$$

$$\theta + \lambda + \mu = 1$$

Where G_i is score of i_{st} endpoint. Especially, selection is performed independently between high-priority and low-priority groups, if there is no endpoint in expected priority groups, we make the selection among other priority groups, if there are one or two endpoints, we simply choose all of them as candidates without picking more endpoints from other priority groups. After grading candidates, we have a highest-scored endpoint, a middle-scored and a lowest-scored endpoint. For candidate set with one endpoint, the highest-scored, the middle-scored and the lowest-scored endpoint are the same one. For candidate set with two endpoint, the middle-scored and the lowest-scored endpoint are the same one. Then we define priority of request: we regard requests whose requirement level ranks high in at least two factor as high-priority requests, we regard the others as low-priority requests. Based on request priority, we have different values for θ, λ, μ, and different selecting strategy. Table 3 describes the definition of request priority. Table 4 gives a hint to select candidate for different request priority. High-priority requests consume more resource than others, so we choose the highest-scored candidate for them. As for low-priority requests, we choose middle-scored one instead. In this way, we use middle-scored candidates which are healthy enough to handle low-priority requests to reduce workload of high-scored candidates to prevent the situation that high-scored candidate become overhead quickly because requests of all priority rush to them.

3. Rewrite destination of request with the selected endpoint IP and dispatch request.

Type	CPU	Memory	Network	Priority
request1	High	High	High	High
request2	High	High	Low	High
request3	High	Low	High	High
request4	High	Low	Low	Low
request5	Low	High	High	High
request6	Low	High	Low	Low
request7	Low	Low	High	Low
request8	Low	Low	Low	Low

Table 3: request priority

Type	θ	λ	μ	Selected endpoint
request1	0.4	0.3	0.3	highest-scored
request2	0.4	0.4	0.2	highest-scored
request3	0.4	0.2	0.4	highest-scored
request4	0.6	0.2	0.2	middle-scored
request5	0.2	0.4	0.4	highest-scored
request6	0.2	0.6	0.2	middle-scored
request7	0.2	0.2	0.6	middle-scored
request8	0.4	0.3	0.3	middle-scored

Table 4: selecting strategy

It is worth mentioning that the algorithm is based on dynamic data provided by load monitor except for request data. The resource consumption, which is a static and stable property of request, is seemed to remain almost the same for a rather long time, so we just evaluate it for once in load monitor. As for dynamic system load of nodes and endpoints, load monitor keeps on collecting load all the time and reports them to loadbalancer at intervals.Loadbalancer makes load-balancing decision based on the newest verison of data it received from load monitor until the next interval arrives. The length of interval has a great influence on the overhead and performance of loadbalancer. Whenever a new interval time arrives, loadbalancer has to recompute the endpoint load to get real-time load so with the increasement of interval, it costs less overhead to compute but has worse performance because the load are not real-time enough. We set the interval to 5 second to make a trade-off between performance and overhead.

4 Evaluation

This section presents a set of evaluations to show the performance of dynamic solution compared with the original ingress controller. We establish a kubernetes cluster to be the base of the whole evaluation. Then we emulate five workload patterns and run them for twice, one with

the original ingress controller equiped, the other with our modified ingress controller equiped to compare their performance. In summary, our results show that the modified ingress controller achieves better performance in most web cases. For independent web applications, the modified ingress controller can balance load well even when there is a great difference between resource consumption of requests. Furthermore, in scaling case, modified ingress controller succeeds to split more load to the new idle endpoints, helping reduce pressure of high-load endpoints and improving RPS. For dependent web applications, modified ingress controller helps protecting endpoints from being influenced by node load, resource race against other endpoints and dependencies to other applications. Modified ingress controller addresses these cases with extra 1/3 overhead. The original ingress controller consumes about 32 35 millicore to synchronize endpoints changes while modified ingress controller consumes 10 more millicore for load monitor and dynamic load balancing.

4.1 Configuration

Our evaluate platform consist of a tsung server, a original ingress controller, our modified ingress controller, several test applications and datebases. All these are deployed as pods on kubernetes. Our kubernetes cluster is made up of a master and five nodes. The hardware configuration of master is Intel Core processor i7 6700 with 4 CPU cores(3.4Ghz), 64GB system memory and a 1TB HDD disk. Nodes share the same configuration: Intel Xeon processor E5 2620 with 8 cores (2.10Ghz), 64GB system memory and a 1TB HDD disk. Many system components are deployed on the master, because it is responsible for managing the whole cluster, making it in high-load state, so we do not put any workload on master. The tsung server is a request generator to emulate workload pattern for comparing performance, and we deploy it alone on node1 without putting any other application on node1 to make sure its perfomance on generating requests is not influenced by other unrelated factors during the evaluation. Similarly, to gurantee the pure performance, we let ingress controller deployed alone on node2. The test application is a java web application connected to a database, it has several apis exposed, requests of different api have diverse consumption on CPU,memory,network and database. The database is a MySQL instance accessed by the test applications. We deploy test applications on node3, node4, node5 and database on node6. With different strategies to generate workload, different number and placement of test application and database, we emulate fix workload pattern to compare performance.

4.2 Workload Pattern

In this section, we evaluate four workload patterns to evaluate performance of modified ingress controller compared with original ingress controller. The workload patterns include initialization pattern, scaling pattern, node-blocking pattern and database-blocking pattern with four different setting of endpoint placement, scaling strategies and workload composition.

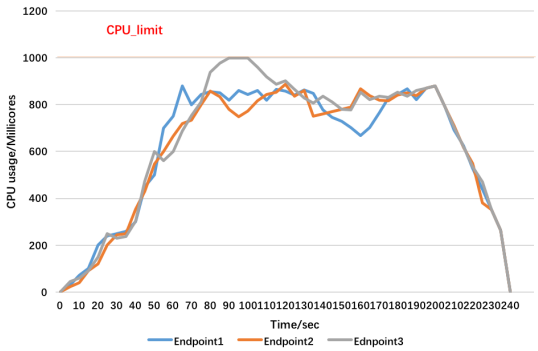

Figure 5: original controller under initialization pattern

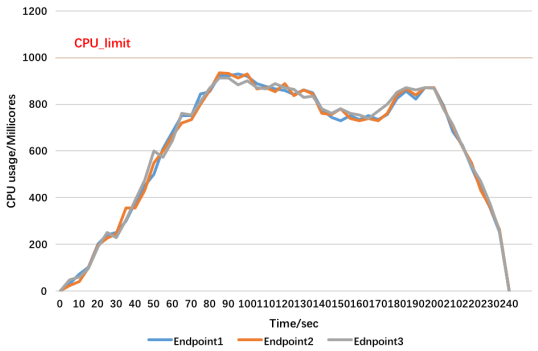

Figure 6: modified controller under initialization pattern

Initialization pattern stands for the initial state when test applications are just deployed: we first deploy three replicas of test application, that is endpoint1 in node3, endpoint2 in node4 and endpoint3 in node5 to prevent that they are impacted by each other. After the deployment, we generate requests of all kinds at a fixed speed by tsung server to access test applications through ingress controller. For convenience, we use the cpu statistics to present performance. Figure 5 presents the performance of original ingress controller under initialization pattern and figure 6 shows the performance of modified ingress controller. Result shows that for the original controller, load of endpoints become unstable and endpoint3 once becomes overhead(its cpu usage exceed its cpu limit) during 80s-100s, causing several re-

quests return 503 to tsung server, it confirms that the default round-robin algorithm only performs well when resource consumption of requests are almost the same, which does not match the real circumstance initialization pattern describes: different kinds of requests have diverse resource requirements, under this circumstance, if we dispatch requests evenly to endpoints, some endpoints can become overhead presumably. Therefore, 503-request appears in evaluation for original ingress controller. Figure 6 shows that with the effort of modified ingress controller, there is no more overhead endpoint, load of endpoints are more even instead. More and more, table 5 points out that original ingress controller finishes 8892 reqeust, with 1108 503-requests, the modified ingress controller finishes 10000 requests with no 503-request, and the total number of requests is 10000.

	Finished	503	Total
Original	8892	1102	10000
Modified	10000	0	10000

Table 5: Request summary of initialization pattern

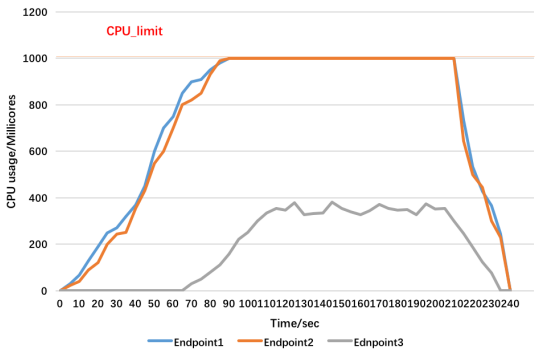

Figure 7: original controller under scaling pattern

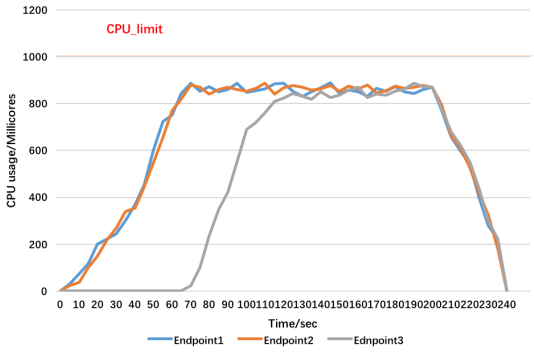

Figure 8: modified controller under scaling pattern

Scaling pattern happens when administrator decides to scale up test applications because the workload is too large for current number of endpoint to handle. To emulate this pattern, we first deploy endpoint1 in node3 and endpoint2 in node3 and generate high workload to keep these two endpoints in high-loaded state but not overload. Then we deploy endpoint3 in node5 and increase the workload to the level that can keep three endpoints in high-loaded state ideally. Figure 7 presents the performance of original ingress controller under scaling pattern and figure 8 shows the performance of modified ingress controller. Result shows that that when we increse workload at 65s, original ingress controller failes to dispatch appropriate workload to endpoint3, which should be assigned more workload to help endpoint1 and endpoint2. On the contrary, the modified ingress controller succeeds to lead more workload to endpint3, as a result, no endpoint becomes overhead. Table 6 points out that original ingress finishes 5844 requests, with 2956 503-request while the modified ingress controller finishes 8800 request with no 503-request, and the total number of reqeust is xxx.

	Finished	503	Total
Original	5844	2956	8800
Modified	8800	0	8800

Table 6: Request summary of scaling pattern

Node-blocking pattern is established to evaluate performance of ingress controllers when their hosts are in high-loaded state. Firstly, we deploy endpoint1 in node3 and endpoint2 in node4. Then we generate workload for node3 to keep it on high-load state, leaving only enough resource for endpoint1. After that, we generate high workload to keep these two endpoints in high-loaded state but not overload and by comparing the RPS, we obtain performing disparity between original ingress controller and modified ingress controller. Through the comparison between figure 9 and figure 10, we find that it costs the original one about 260 second to handler 10000 requests while it costs the modified one only 250 seconds, which demonstrates the fact that load of host can affect the performance of endpoints. That is because the default isolation mechanism of docker, which is the container engine of kubernetes, can not support node-blocking pattern well. Kubernetes provides *resource.request* and *resource.limit* for users to allocate resource for their applications, however it does not gurantee the monopolization of resource. For example, the request limit and request of cpu are implemented by a combination of *cgroup* commands: cfs_quota_us,cfs_period_us and *cpushare*. These com-

mands insulates cpu through restricting the length of cpu time slice without binding applications to specific cpus, so when preemption happens, the inevitable context switch brings extra overhead. The modified ingress controller reduce context switch by taking load of host into consideration when balancing load so it gets a higher RPS.

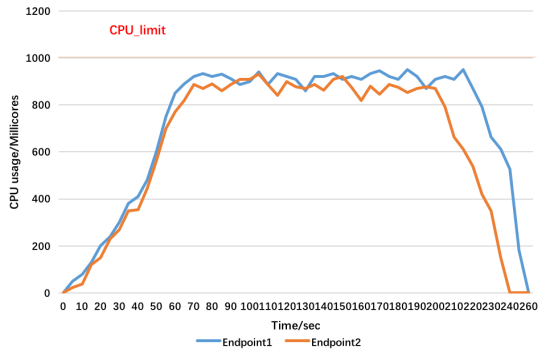

Figure 9: original controller under node-blocking pattern

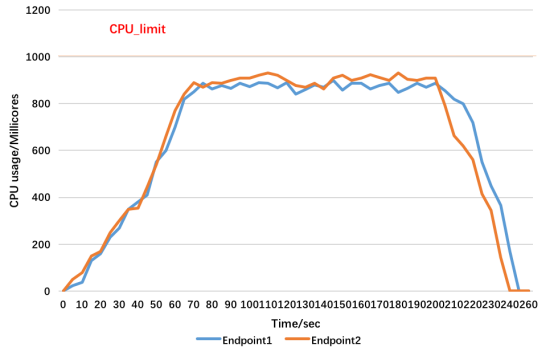

Figure 10: modified controller under node-blocking pattern

Interrelationship pattern describes situation when several endpoints are deployed in the same node, we set up this pattern to figure out the ability of ingress controller to eliminate interrelationship between these endpoints. To emulate this pattern, we put endpoint1, endpoint2 in node3, and endpoint3 in node4. Then we generate high workload to keep these endpoints in high-loaded state. Figure 11 shows that in original case, loads of endpoint1, endpoint2 are higher than other endpoints at the same time, and finally it finishes all requests in 245 seconds. That is because endpoints of the same application share the same resource feature, so if endpoints are deployed into the same node, their race for resource becomes fierce, thus they impact on performance of each other. Figure **??** demonstrates that the modified ingress controller succeeds to reduce race and finishes all requests in 240 seconds.

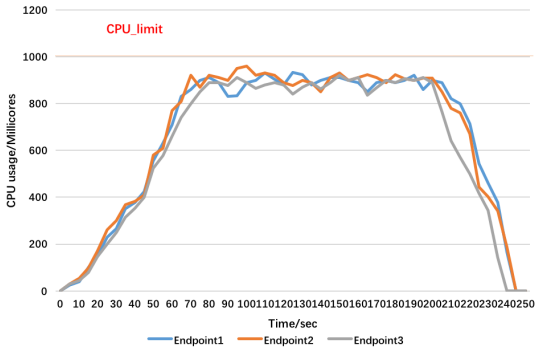

Figure 11: original controller under interrelationship pattern

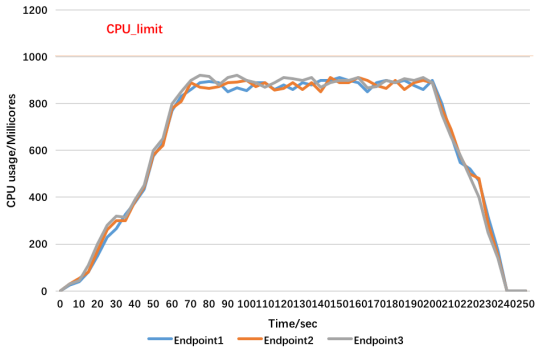

Figure 12: modified controller under interrelationship pattern

Database-blocking pattern aims to figure out whether ingress controller can perform well under the micro service background. Under this circumstance, web applications tend to rely on each other rather than finish jobs independently, so QOS is not only decided by application itself, but also other applications it relies on. For convenience, we choose database-accessing applications as study case and set up database-blocking pattern: First we deploy endpoint1 in node3, endpoint2 in node4, endpoint3 in node5, database1 and database2 in node6, then we configure endpoint1, endpoint2 to connect to database1 and endpoint3 to database2. Finally, we generate workload to keep these endpoints in high-loaded state but not overhead. By comparing RPS, we get their performing disparity. Figure 13 shows that original ingress controller failes to divide load of database1 to datebase2 which comes from endpoint1 and endpoint2. As a result, database1 bears more load than database2 all the time, pulling down the entire RPS. As figure 14 shown, modified ingress controller succeeds to balance load across database1 and database2, finishing all requests in 240

seconds.

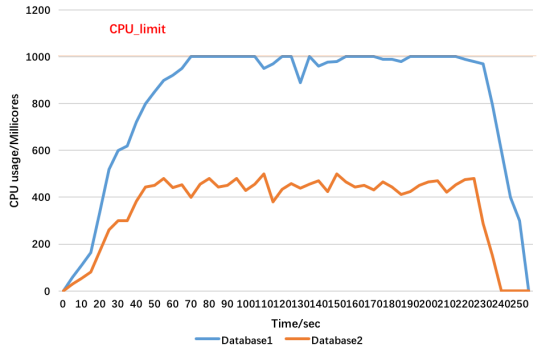

Figure 13: original controller under node-blocking pattern

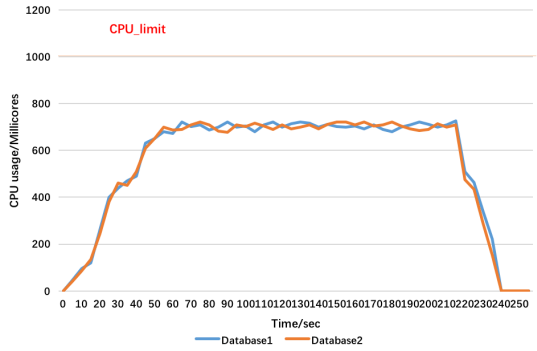

Figure 14: modified controller under node-blocking pattern

5 Related Work

5.1 Load balancing classification

According to R Kaur [7], load balancing algorithms can be mainly divided into two types: static load balancing algorithm and dynamic load balancing algorithm. Furthermore, dynamic algorithms have two types, they are distributed and non-distributed approach. The biggest difference between static and dynamic algorithms is whether they depend on server load or request property. A request can either access a file or require processing data. Some researches have been done to reduce latency to access static files: S Ayyasamy [1] concentrates on static resource placement in P2P content distribution. According to acccess frequency, static resources are divided into hot and cold resources. Data transmission performance is enhanced by placing hot resources to server with stronger processing ability and lower access latency. G Park adopts choice algorithm to optimize file replication and data partition policies. Q Huang [6] uses hashing and cache to migrate load imbalance. other researches focus on data-processing requests: Sharma [9] performs dynamic load balancing under the assumption that the resource consumption of request have already been known as job metric, he measures servers by several metrics like ability metric and load metric. With the combination of job metric and server metrics, he evens load across servers. Chandak [3] divides servers into three levels: lightly loaded, moderately loaded and heavily loaded accoring to their cpu usage. In his research, moderately loaded is regarded as the stable and final level, he performs load balancing by dispatching more workload to lightly loaded server and less workload to heavily loaded servers. L Zhang [13] describes processing time of a request with cpu processing time, memory usage and disk access time. For each server, he maintains a mission list to record requests the server is processing. When a new request arrives, if the processing ability of servers is higher than the requirement sum of the new request and requests in mission list, they are regarded suitable to process the request. Z Xu [12] and ME Gebrehiwot [5] adjust weight of servers in Round-robin and least connection algorithm according to their ability. G Velusamy [10] defines server affinity to adjust load balancing algorithm. The server affinity presents whether a server can handler a specific request type faster than other server. Loadbalancer collects and analyzes logs from servers all the time to get real-time server affinity, with which it dispatches requests to suitable servers. V Cardellini [2] proposes the adaptive TTL algorithm, which adds geographic information and server load to the Round-robin algorithm. The adaptive TTL algorithm tends to dispatch requests to low-loaded servers that are geographically close to requests senders.Unlike the non-distributed approaches mentioned above, Eludiora [4] and Menon [8] adopt distributed solutions. Eludiora's research regulates jobs/tasks migration to minimize extra bandwidth consumption, and therefore improve RPS. Menon establishes a broadcast protocol for server, for each low-loaded server, it broadcast its ip to two random servers, if these servers are overload, they transfer new requests to the low-loaded servers. Unfortunately, there is no effective solution for load balancing of container-based cluster. For container-based cluster like Kubernetes, containers generally own the same geographical position and the same process ability. The key factors that cause load imbalance are the resource consumption of request and dependency between services. So this paper proposes a load monitor to learn resource consumption and a more thorough load metric for server, which adopt service dependency.

5.2 Load balancing metrics

Metrics are needed to evaluate performance of a load balancing algorithm. R Kaur [7] summarys that throughput is calculated to the ability of algorithm to process as many requests as possible during a fixed period. The performance of algorithm is high if its throughput is high. Response time is the time that are taken to process a request. With a lower response time, a algorithm acheives better performance. This paper uses RPS to cover throughput and response time, a higher RPS means that requests are led to more suitable servers, thus bringing lowwer response time and higher throughput. For distributed algorithm, migration time costed by migrating request from one server to another is calculated to evaluate performance. Apart from RPS, this paper proposes request faults. If a algorithm failes to balance load, overload servers may failes to process requests, cauing 503 Service Unavailable. Through counting number of 503-request, this paper knows ability of load balancing algorithm to balance load when the cluster are in high-loaded partially.

6 Conclusion and Future Work

With the widespread of internet and connected devices, web services are growing in an explosive scale. For users who deploy their web services into kubernetes built in bare machine or private cloud environments,the default ingress controller is not capable of adjusting its load balancing strategy with the demands of web services. That is, the great difference between resource consumption of APIs and the dependency between services require a more fine-grained load balancing algorithm and the scaling of web services requires load balancer to drive more traffic to new idle server in case appearance of 503 Service unavailable. To address these problems, this paper provides a dynamic load balancing solution for kubernetes. The solution includes a load monitor, which can learn resource consumption and collect pod load, and a dynamic load balancing algorithm. The dynamic load balancing algorithm combines system load and service dependency to describe pod load, and finally select appropriate pod according to different resource consumption.

For future work, we will improve the scalability and reliability of ingress controller. As we know, the ingress controller works as the proxy of services alone in kubernetes, once it goes wrong, the web services become unavailable. We plan to set up monitoring for ingress controller, if something goes wrong with it, a backup ingress controller will take place of it to work. If the workload is beyond the ability of current ingress controller, more ingress controllers should be started to spread the load, more and more, all these ingress controllers should present as the same user-transparent proxy. We hope this solution help ingress controller perform better in kubernetes.

7 Acknowledgements

.....We thank our shepherd Dan Tsafrir, Haibo Chen, and the anonymous reviewers for their insightful comments. This work was supported by National Science and Technology Major Project (No. 2013ZX03002004), National R&D Infrastructure and Facility Development Program (No. 2013FY111900), NRF Singapore CREATE Program E2S2, and Shanghai Key Laboratory of Scalable Computing and Systems. Prof. Haibing Guan is the corresponding author.

References

[1] AYYASAMY, S., AND SIVANANDAM, S. N. A cluster based replication architecture for load balancing in peer-to-peer content distribution. *International Journal of Computer Networks Communications 2*, 5 (2010).

[2] CARDELLINI, V., COLAJANNI, M., AND YU, P. S. Dynamic load balancing on web-server systems. *Internet Computing IEEE 3*, 3 (2015), 28–39.

[3] CHANDAK, A., JAJU, K., KANFADE, A., LOHIYA, P., AND JOSHI, A. Dynamic load balancing of virtual machines using qemu-kvm. *International Journal of Computer Applications* (2012).

[4] ELUDIORA, S., ABIONA, O., ADEROUNMU, G., OLUWATOPE, A., ONIME, C., AND KEHINDE, L. A load balancing policy for distributed web service. *International Journal of Communications Network System Sciences 3*, 8 (2010), 645–654.

[5] GEBREHIWOT, M. E., AALTO, S., AND LASSILA, P. Energy efficient load balancing in web server clusters. In *International Teletraffic Congress* (2017), pp. 13–18.

[6] HUANG, Q., GUDMUNDSDOTTIR, H., VIGFUSSON, Y., FREEDMAN, D. A., BIRMAN, K., AND RENESSE, R. V. Characterizing load imbalance in real-world networked caches. In *ACM Workshop on Hot Topics in Networks* (2014), p. 8.

[7] KAUR, R., AND LUTHRA, P. Load balancing in cloud computing. *Int.conf.on Recent Trends in Information Telecommunication Computing Itc* (2014).

[8] MENON, H., AND KAL, L. *A distributed dynamic load balancer for iterative applications*. IEEE, 2013.

[9] SHARMA, D., AND SAXENA, A. B. Framework to solve load balancing problem in heterogeneous web servers. *International Journal of Computer Science Engineering Survey 2*, 1 (2011).

[10] VELUSAMY, G., AND LENT, R. Smart load-balancer for web applications. In *International Conference* (2017), pp. 19–26.

[11] WILKES, J., WILKES, J., WILKES, J., WILKES, J., AND WILKES, J. Borg, omega, and kubernetes. *Communications of the Acm 59*, 5 (2016), 50–57.

[12] XU, Z., AND WANG, X. A modified round-robin load-balancing algorithm for cluster-based web servers. In *Control Conference* (2014), pp. 3580–3584.

[13] ZHANG, L., LI, X. P., AND YUAN, S. A content-based dynamic load-balancing algorithm for heterogeneous web server cluster. *Computer Science Information Systems 7*, 1 (2010), 153–162.

[14] ZHAO, D., MOHAMED, M., AND LUDWIG, H. Locality-aware scheduling for containers in cloud computing. *IEEE Transactions on Cloud Computing PP*, 99 (2018), 1–1.