A Container Service Chain Placement Greedy Algorithm Based on Heuristic Information

Qiang Yue\(^1\)*, Xiangtao Liu\(^2\), Lijia Fang\(^1\), Xiaoxiao Wang\(^1\) and Wenbin Hu\(^1\)

\(^1\)SIC Electronic Bill Platform Information Service Co., Ltd., Beijing, China
\(^2\)Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou, China

Email: yueqiang@12366.com

Abstract. Container technology is a light weight back-end virtualization solution which copes with the growing demand for Internet service concurrency. Network services can be provided by a series of containers between which network requests are sent and responded. We call these cooperating containers a container service chain. On a certain physical cluster, there are various available container placement of a service chain. Different container placement causes different service response time. In order to reduce the response time of container service chain, a greedy algorithm for container service chain placement is proposed based on heuristic information. Through quantitative analyzing resources provided by hosts and consumed by containers, we build a mathematic model of container service chain. We use this model to decide where to place container when service is expanded, and where to remove container when service is reduced. Our model reduced the response time of container service chains by 40% compared to average placement.

Keywords. Container placement; performance modelling; container networks; I/O response time.

1. Introduction

In the past few decades, the scale of Internet was growing rapidly. In the 44th China Statistical Report on Internet Development publish by CNNIC in 2019, the number of Chinese netizens has increased to 854 million [1]. The huge network user crowd are consuming the Internet through various services, such as websites, mobile applications, and online games. Take the example of T-mall at double eleven, which is a shopping festival of china, the peak request concurrency is 140 thousand per second in 2015, and 175 thousand per second in 2016.

Therefore, the back-end service providers are focusing on providing the services at scale. Virtualization is one of the emerging technologies used in data centers and cloud environments to improve both hardware and development efficiency. In the past, the virtual machine is a common virtualization method. A virtual machine has its own isolated CPU, memory, block I/O, and network resources, etc. However, using virtual machines means running many duplicate instances of the same OS and many redundant boot volumes. They make virtual machines suffer from noticeable performance overhead, large storage requirement, and limited scalability.

As a result, containers were designed and implemented to address these limitations. Containers can provide multiple isolated service units of the application while sharing the host operating system and physical resources. Felter et al. [2] compared the resource consumption between containers and virtual machines under the same deployment, and draw the conclusion that containers cost less resource.
Containers have three stages of development. In the beginning, a container is only a calculating thread running on a Unix server. Then comes docker, which introduces a way of simplifying the tooling required to create and manage containers. Docker also uses cgroup to limit the resource that can be used by a container, which gives containers certain independence. It means seeing from the physical server, containers are essentially just regular processes. Meanwhile, seeing from the container, a container enjoys a virtualized resource environment, not only just CPU and memory, but also bandwidth, ports, disk i/o, etc. The latest stage is container cluster orchestrator, for example, Google Kubernetes and Docker Swarmkit. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. The prototype of Kubernetes is Borg, a Google internal use container control system, and Google published it at 2014. At present, Kubernetes becomes the most mainstream container cluster orchestrator.

An essential feature of Kubernetes is automatically changing the scale of the container cluster based on the request load. Kubernetes proposes a concept called pod. A pod consists of one or more containers of a same image. A service provided by a Kubernetes cluster consists of one or more pods. When the request number of a service changes, Kubernetes will automatically increase or decrease the container number of the pods. The relationship between Kubernetes service, pod and container is shown in figure 1.

![Image](image.png)

**Figure 1.** The relationship of K8s service, pods and containers.

There are many studies about how to improve the performance of virtual backend. Harter et al. [3] find that the container startup latency is considerably larger than expected, in which copying package data accounts for most of container startup time. So, they implemented a tool called slack which can reduce the amount of copying and transferring packages to speed up the startup procedure. Nathan et al. [4] proposed CoMICOn, which addresses the problem by sharing the image in a cooperative manner. Hegde et al. [5] also proposed SCoPe to characterize the provisioning time in terms of system features. It presents a statistical model to guide provisioning strategy. However, these works are all aiming at reducing the startup time, and ignored the service runtime response time.

Assuming that the hardware of physical server cluster, the configuration and amount of containers are determined, the runtime response time can only be affected by the location of containers, in other words, the placement of containers. It is different from placement problem with topology like middlebox placement [6-9]. Gog et al. [10] compared the scalability between Docker Swarmkit [11] and Google Kubernetes [12], and then proposed firmament to achieve low latency in large-scale clusters by using multiple min-cost max-flow algorithms. On the other hand, focusing on workload scheduling, Kaewkasi et al. [13] describes an Ant Colony Optimization algorithm for a cluster of Docker containers. However, the algorithm does not distinguish various containers, which usually have a diverse requirements. Mao et al. [14] proposed DRAPS to reduce the physical resource cost rate instead of response time. What’s more, the studies we mentioned above are all general solutions. Our work, on the contrary, aims at find an optimal placement of container service chain scaling.

The way in which Kubernetes scales the cluster is simply equal distribution. It can be concluded as adding in turn and deleting randomly. This naïve way is usually unable to find the optimal deployment.
If the connection relationship between the pods is determined, in other words, service chain exists, we can use the relationship as heuristic information to improve the performance of the service chain. It will reduce the total response time of the cluster.

In this paper, we make the following contributions:

- **A Service Chain Model:** We propose a model to simulate the response procedure of container cluster when the hardware of physical server cluster, the configuration and number of containers are given. By this model we can calculate the response time of the service provided by the cluster.

- **A Greedy Optimization Algorithm:** We propose a complete container placement scheme based on greedy algorithm, which assigns the container to appropriate physical node and reduce service response time in a service chain cluster. The simulation demonstrates that our greedy algorithm outperforms the Kubernetes default average scheme and reduces response time as much as about 40% on a given cluster.

The rest of this paper is organized as follows. Section II describes the service chain model, and the way to calculate the response time of a service chain. Section III elaborates the greedy algorithm to find the optimal placement. The algorithm is implemented in Section IV. Section V shows the evaluation results of the greedy algorithm. Section VI concludes this paper and shows the future work.

## 2. Models

In this section, we will design a model to imitate the response procedure of container service chain, and explain how to calculate the response time.

### 2.1. Structure of Container Service Chain Cluster

The structure of a typical container service chain is shown in figure 2. Physical servers provide various resources, such as CPU, bandwidth and store. The container service chain in figure 2 contains three parts. The first part is DB, which stores data and provide data to Web server. Therefore, this part requires store resource mostly as well as network resource. The second part is Web, which need to handle the data and output the web page to LB. The mostly cost resource is CPU. The last part LB directly communicate with user requests of high concurrency, so it need network resource mostly. We describe the relationship between containers by the distance to users. In the example above, DB is the furthest part. Meanwhile, LB is the nearest part.

![Figure 2. A typical K8s service chain.](image)

In Kubernetes, the whole container service chain can be deployed as a service. Each part of the service chain is a pod. There is data transmission between two adjacent parts. Different replicas of a pod can be deployed on different physical servers.

### 2.2. Constraints of Container Deployment

To answer the question that how many containers we need to deploy, we suppose that a unit of requests need a unit of service chain, and one-unit service chain consists of one replica of each pod.

To simplify the problem, we suppose physical servers only provides three kinds of resources which we mentioned above. CPU resource represents resources for calculation, such as CPU cycle and RAM. Bandwidth resource represents network resources. Bandwidth is consumed when two adjacent pods in a
service chain are placed on different physical servers. Store resource represents the storage space provided by hard disks.

We define a placement plan as how we put a container service chain cluster on a physical server cluster when both of them have fixed parameters and amount. These three constraints work in different mechanisms. Store constraint requires that the summation of storage space needed by pods on a single physical server cannot be more than the storage space provided by it. However, there are no such simply sum-up requirement on CPU and bandwidth constraints. Limited calculate resource means that calculate tasks are solved in a fixed speed. If we put too many calculate tasks on a physical server, then this server needs a lot of time to solve them, which will become a bottleneck of the whole cluster. Bandwidth constraint behaves similar with CPU constraint. Placement does not change the total calculate task, but changes the total data transmission. We will explain how to calculate the response time of a placement in next subsection.

2.3. Calculation of Response Time

In figure 3, there are 3 units of requests, and the required 3 units of container service chains are deployed on 3 physical servers. Remember that a unit of requests is not a single request, but a number of requests that need a unit of service chain to handle. In this example, when a user sends a request to the server cluster, the request first arrives at the LB pod. Then LB pod send calculate request to the Web pod. The Web pod asks DB pod for data. After fetched data form DB pod, Web pod start to calculate and send the result to the LB pod. Finally, the LB pod send the web page to the user. A request finished. In this example, there are only data transmission from a far pod to a near pod because the data transmitted form a near pod to a far pod can be ignored. However, in general scenarios, data transmission in both directions exists.

![Figure 3. An example of incrementally scaling K8s cluster.](image)

At the micro level, containers in a service chain work in order to response a request. However, at the macro level, a unit of requests contains massive requests. When a part of a request is solved on a part of the service chain, other parts of the service chain are handling other requests. Therefore, the response procedure is non-sequential in time, but a concurrency procedure.
The response time of the cluster is decided by the physical server with longest response time, so we first calculate response time separately for each physical server and then compare them. The response time of a physical server contains three parts, the transmission time of ingress traffic, the transmission time of egress traffic and the calculate time. The response time is decided by the longest one of them.

The transmission time of ingress traffic and egress traffic can be calculated as equations (1) and (2). $T_{input}$ and $T_{output}$ represent transmission time of ingress traffic and egress traffic. $D_{input}$ and $D_{output}$ represent the ingress and egress traffic of each pod on a physical server. $BW_{input}$ and $BW_{output}$ represent the ingress and egress bandwidth of the physical server. Generally speaking, the ingress bandwidth of a NIC equals the egress bandwidth of it. Suppose the bandwidth is known, we need to calculate the ingress traffic and egress traffic in order to calculate the transmission time.

$$T_{input} = \frac{\sum D_{input}}{BW_{input}}$$  

$$T_{output} = \frac{\sum D_{output}}{BW_{output}}$$  

Considering the load balance mechanism between adjacent pods in the service chain, the ingress or egress traffic of a pod comes from or goes to every adjacent pod equally. Suppose that the ingress traffic comes from the far pods, and the egress traffic comes from the near pods, the way to calculate $D_{input}$ and $D_{output}$ is shown as equations (3) and (4). $D_{far}$ and $D_{near}$ represent the traffic generated by a unit of this pod. $C_{far}$ and $C_{near}$ represent the all far pods and near pods, while $C_{far\text{diff}}$ and $C_{near\text{diff}}$ represent far pods and near pods deployed on different physical servers.

$$D_{input} = D_{far} \times \left( \frac{\sum C_{far\text{diff}}}{\sum C_{far}} \right)$$  

$$D_{output} = D_{near} \times \left( \frac{\sum C_{near\text{diff}}}{\sum C_{near}} \right)$$

Assuming that there are $m$ units of service chains, for each pod, we need to calculate the traffic between it and $2m$ adjacent pods. The time complexity is shown in equation (5).

$$T(n) = O(n^2)$$  

Compared to transmission time, it is easier to calculate the computing time. It can be calculated as dividing the summation of computing tasks deployed on a physical server by its computing power. The process is shown in equation (6). $T_{cal}$ represents the computing time. $C_{cal}$ represents the computing power.

$$T_{cal} = \frac{\sum D_{cal}}{C_{cal}}$$

We only need to traverse each pod once to calculate the computing time, so the time complexity is shown in equation (7). Because it’s one magnitude lower than calculating transmission time, the time complexity to calculate response time of a physical server can be represented by equation (5).

$$T(n) = O(n)$$

Here comes an example showing how to calculate the response time of a container service chain. The placement is shown in the top part of figure 3. The resources provided by physical server and the tasks of each pod are shown in table 1. For server 0, the total computing task is:

$$50 \times 2 + 1000 + 200 = 1300$$

| Hardware | Parameter |
|----------|-----------|
| CPU      | Intel® Core(TM) i5-8250U @ 1.6GHz 1.8GHz |
| RAM      | 8.00 GB   |

**Table 1.** The configuration of the laptop.
The computing time is:

\[ 1300 / 5 = 260 \] (9)

The ingress transmission task is:

\[ 100 \times 2 \times 2/3 + 10 \times 2/3 + 1000 = 1140 \] (10)

The ingress transmission time is:

\[ 1140 / 5 = 228 \] (11)

The egress transmission task is:

\[ 100 \times 1/3 + 10 \times 2 \times 1/3 = 40 \] (12)

The egress transmission time is:

\[ 40 / 5 = 8 \] (13)

Therefore, the bottle neck of server 0 is computing time, 260, which is the response time of server 0. The response time of server 1 and server 2 can also be calculated in the same way. The response time of server 1 is 250, and server 2 is 240. As a result, the response time of the whole service chain is the maximum of them, 260.

3. Algorithm

In this section, we will design a greedy algorithm to find the optimal container service chain with minimal response time.

3.1. Analysis of Simple Traverse Algorithm

With the response time computing model, we can calculate the time complexity of searching the optimal placement.

If we use simple traversal algorithm to search the optimal placement, we need to calculate response time of every possible placement. Suppose there are x physical servers with the same parameters. A service chain contains y pods, and there are z units of requests. Without regard to the storage constraint, the number of possible placements can be calculated as a combination problem shown in equation (14).

\[ N = \binom{x-1}{y} \times \binom{yz \times (x-1)}{y} \times \binom{yz \times (x-1)!}{y!} \times \binom{(y+z)!}{y!} \] (14)

The time complexity exceeds polynomial complexity. Therefore, finding optimal placement with simple traversal algorithm is a NP-hard problem.

What’s more, even if we can calculate the optimal placement under a given request number, the request number changes rapidly in an auto scaling Kubernetes cluster. The optimal placement under a request number may be unable to adjust to the optimal placement under another request number by simply adding and removing pods.

Still in the situation in figure 3, the optimal placement under 3 requests is shown as above, with a response time of 260. The optimal placement under 4 requests is shown in right bottom with a response time of 420. However, the optimal placement without pod migration is shown in left bottom with a response time of 440.

In conclusion, calculating the optimal placement under a given request number is complex and meaningless. What we need to solve is the problem that when the request number increases or decrease, where should we deploy new pods or which pods should we remove, in other words, an incremental optimization algorithm of pod placement.

3.2. Greedy Algorithm

We simply use a greedy algorithm to optimize service chain incremental placement.
A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time. The problem of place service chain on physical servers can be easily decomposed into a series of single service chain deployment problem or single pod deployment problem.

There are two optional greedy algorithms. We describe the problem of finding optimal service chain placement as follows. There are x physical servers. A unit of service chain contains y pods. An increasement of z units of service chain takes place in a modification.

The first algorithm is finding the optimal placement in the granularity of service chain. In the procedure we add a unit of service chain to the cluster, we traverse all possible plans of placing the y pods in a unit of service chain on x physical servers and find the placement with the shortest response time. Repeat this process z times, and we will find the optimal placement. The number of possible placements can be calculated as equation (15).

\[ N_{\text{incre}} = z \times x^y \] (15)

Although x and y are both constant, and there is a linear relation between the service chain increase number z and the number of possible placements, for long service chain and large-scale physical server cluster, \( x^y \) will becomes larger than \( 2^{32} \), which is the order of magnitude that computers can solve. So it is infeasible to find the optimal placement in the granularity of service chain.

The other algorithm is finding the optimal placement in the granularity of pod. In the procedure we add a unit of service chain to the cluster, for each pod we traverse all x physical servers to find an optimal place which makes the response time shortest. Each of the y pods in a service chain need to be placed, and there are z units of service chain need to be added to the cluster. The number of possible placements can be calculated as equation (16).

\[ N_{\text{incre}} = z \times x \times y \] (16)

Because x and y are both constant, this algorithm has linear time complexity. Besides, \( x^y \) is far less than \( 2^{32} \), so this algorithm is feasible.

\[ P_a = \frac{1+\alpha}{1+\alpha+D} \] (17)

4. Implementation

However, in the second greedy algorithm, the pods in a unit of service chain is placed one by one, there is the problem that the numbers of adjacent pods are not equal, which means it is unable to calculate the response time of the cluster. To solve this question, we make the following improvements.

First, use the actual number of pods to calculate the data transmission between pods. If a pod has 3 far pods and 4 near pods, the data transmitted to far pods are divided into three equal parts and that transmitted to near pods are divided into four equal parts.

Second, imitate the random choosing and iterating idea of K-means algorithm [15]. After find the greedy optimal solution by the first traverse, we relocate the pods in this unit of service chain one by one. This procedure repeats several times.

Third, that there are multiple optimal solutions in some situations. We find that a balanced placement will benefit the subsequent optimization procedure. Therefore, when two placements have a same response time, the variance of the physical servers’ load is calculated. The load can be defined as the number of pods or data transmission on a physical server, etc. In the algorithm implemented in this paper, the load is computing tasks on a physical server.

Because the reason why the numbers of adjacent pods are not equal is the pods in a unit of service chain is deployed one by one, the number differnce between adjacent pods is at most 1. When the number of service chain is huge, this can be ignored. By applying improvement 2, the placement of prior pods can be adjusted with the placement of after pods. If we repeat the relocation process in
improvement \(2 \times m\) times, the response time needs to be calculated \(N_{\text{incres}}\) times, which shown in equation (18).

\[
N_{\text{incres}} = z \times (m + 1) \times x \times y
\]  

(18)

In our implementation, \(m\) is 5.

The algorithm of adding a unit of service chain is shown in algorithm 1.

**Algorithm 1 Incremental pod placement algorithm**

**Input:** Physical server array \(S\), service chain \(C\), max repeat time \(m_{\text{max}}\)

**Output:** The Optimal placement

1. **PROCEDURE:** INCREASE\((S, C, M_{\text{max}})\)
2. for \(m=0\) to \(m_{\text{max}}\) do
3.   for \(i=0\) to \(C\).size() do
4.     Record the minimal response time of all tried placement \(m_{\text{time}}=\text{inf}\). The index of placement \(\text{plan}=1\). The variance \(\text{variance}=\text{inf}\).
5.     for \(j=0\) to \(S\).size() do
6.       Place the \(i\)th pod on the \(j\)th server
7.       Calculate the response time \(\text{temp}_{\text{time}}\) and the variance of loads \(\text{temp}_{\text{Var}}\)
8.       if \(\text{temp}_{\text{time}}<\text{mintime}\)
9.         \(\text{mintime} = \text{temp}_{\text{time}}\)
10.        \(\text{plan}=j\)
11.       \(\text{minVar}=\text{temp}_{\text{Var}}\)
12.      else if \(\text{temp}_{\text{time}}=\text{mintime} \&\& \text{temp}_{\text{Var}}<\text{minVar}\)
13.         \(\text{plan}=j\)
14.        \(\text{minVar}=\text{temp}_{\text{Var}}\)
15.      Remove the latest deployment of the \(i\)th pod
16.    Deploy the \(i\)th pod on the \(\text{plan}_{\text{th}}\) physical server
17. return Deployment

If the scale change is decreasing instead of increasing, we can directly remove the latest unit of service chain. The time complexity is just \(y\).

Compared to other placement problem, such as Knapsack problem, service chain placement problem doesn’t have a specific optimization goal. The Goals of different users are also different. Time sensitive service requires that the response time should be as short as possible. Cloud service provider prefers less data transmission. The consumer would like to rest as less cloud servers as possible.

To fit our algorithm with different optimization goals. We only need to modify the comparison in each loop. In this paper, our optimization goal is response time, so we calculate and compare response time in each loop.

5. Evaluation

In this section, we will compare the response time of the placement generated by our greedy algorithm and equally distributed algorithm used by Kubernetes. The parameters in our simulation is shown as follows.

The length of service chain:
(1) Short service chain with 3 pods in length.
(2) Long service chain with 10 pods in length.
(3) Forked service chain with 10 pods.

The scale of physical server cluster:
(1) Small cluster containing 5 servers.
(2) Medium cluster containing 20 servers.
(3) Large cluster containing 100 servers.
Each scene is a combination of a service chain and a physical server cluster. There are 9 scenes in total. The comparison between our greedy algorithm and equally distribution algorithm is shown in figure 4. In the scene with long service chain, the greedy algorithm can reduce about 40% response time with most of the scales. When the length of service chain is short, the greedy algorithm also approximates the performance of equally distribution algorithm.

![Figure 4. Response time of service chains.](image)

The computing time of finding placement is shown in figure 5. The configuration of the laptop we used is shown in table 1. The computing time has the polynomial correlation with the service chain scale, which proves the theoretical result. Even on a laptop, the placement can be calculated in seconds. The cost time can be ignored compared to the start time of the containers.
6. Conclusion
In this work, we built a container service chain model for Kubernetes clusters, and use this model to calculate response time. Then we designed a greedy algorithm to find the optimal placement of service chain on physical servers and implemented it. Compared the equally distribution algorithm used by Kubernetes, our algorithm reduces response time by 40%, and the time complexity is acceptable.

The future work is to build a better model for Kubernetes clusters and abstract out more optimization goals. And find a white box optimization algorithm instead of this black box greedy algorithm.

Acknowledgement
This work is supported by the National Key Research Program of China (Grant NO. 2017YFB0802701, 2018YFB0803901).

References
[1] CNNIC 2019 The 44th China Statistical Report on Internet Development.
[2] Felter W, Ferreira A, Rajamony R and Rubio J 2015 An updated performance comparison of virtual machines and linux containers 2015 IEEE Internation Symposium on Performance Analysis of Systems and Software (ISPASS) pp. 171-172.
[3] Harter T, Salmon B, Liu R, Arpaci-Dusseau A C and Arpaci-Dusseau R H 2016 Slacker: Fast distribution with lazy docker containers 14th USENIX Conference on File and Storage Technologies (FAST’16) pp 181-195.

[4] Nathan S, Ghosh R, Mukherjee T and Narayanan K 2017 Comicon: A co-operative management system for docker container images 2017 IEEE International Conference on Cloud Engineering (IC2E) pp. 116-126.

[5] Hegde A, Ghosh R, Mukherjee T and Sharma V 2016 Scope: A decision system for large scale container provisioning management 2016 IEEE 9th International Conference on Cloud Computing (CLOUD) pp. 220-227.

[6] Kang N, Liu Z, Rexford J and Walker D 2013 Optimizing the one big switch abstraction in software-defined networks Proceedings of the ninth ACM conference on Emerging Networking Experiments and Technologies pp 13-24.

[7] Arashloo M T, Koral Y, Greenberg M, Rexford J and Walker D 2016 SNAP: Stateful network-wide abstractions for packet processing Proceedings of the 2016 ACM SIGCOMM Conference pp 29-43.

[8] Huang W, Lin W and Xu S 2017 Application of graph theory and hybrid GA-SA for operation sequencing in a dynamic workshop environment Computer-Aided Design and Applications 14 (2) 148-159.

[9] Li Q, Jiang Y, Duan P, Xu M and Xiao X 2017 Quokka: Latency-aware middlebox scheduling with dynamic resource allocation Journal of Network and Computer Applications 78 253-266.

[10] Gog I, Schwarzkopf M, Gleave A, Watson R N and Hand S 2016 Firmament: Fast, centralized cluster scheduling at scale 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) pp 99-115.

[11] Docker swarmkit https://github.com/docker/swarmkit.

[12] Bernstein D 2014 Containers and cloud: From lxc to docker to kubernetes IEEE Cloud Computing 1 (3) 81-84.

[13] Kaewkasi C and Chuenmuneewong K 2017 Improvement of container scheduling for docker using ant colony optimization 2017 9th International Conference on Knowledge and Smart Technology (KST) pp 254-259.

[14] Mao Y, Oak J, Pompili A, Beer D, Han T and Hu P 2017 Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster 2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC) pp 1-8.

[15] Hartigan J A and Wong M A 1979 Algorithm AS 136: A k-means clustering algorithm Journal of the Royal Statistical Society. Series C (Applied Statistics) 28 (1) 100-108.