CNTC: A Container Aware Network Traffic Control Framework

Lin Gu[1], Junjian Guan[1], Song Wu[1](✉), Hai Jin[1],
Jia Rao[2], Kun Suo[2], and Deze Zeng[3]

[1] SCTS/CGCL, Huazhong University of Science and Technology, China
{lingu,junjianguan,wusong,hjin}@hust.edu.cn
[2] The University of Texas at Arlington, USA
{jia.rao,kun.suo}@uta.edu
[3] University of Geosciences, China
dazzae@gmail.com

Abstract. As a lightweight virtualization technology, containers are attracting much attention and widely deployed in the cloud data centers. To provide consistent and reliable performance, cloud providers should ensure the resource isolation since each host consists of multiple containers sharing the host kernel. As a mainstream container system, Docker uses CGroup to provide CPU, memory, and disk resource isolation. Unfortunately, all present solutions ignore the network resource, leading to the resource competition and violating the performance of networked hosts. Although several researches discuss the possibility of leveraging Linux Traffic Control (TC) module to guarantee network bandwidth, they fail to capture the diversity and dynamics of container network resource demands and therefore cannot be applied to container-level network traffic control. In this paper, we propose a Container Network Traffic Control (CNTC) framework which can provide strong isolation and container-level management for network resource with joint consideration of container characteristics and quality of service. To simplify the traffic control, we also design a series of APIs which allow inexpert programmers to perform complicated traffic control on each container. Through experiment results, we show that CNTC works well in all network modes of containers.

Keywords: Container Network · Traffic Control · Network Isolation.

1 Introduction

Recently, container, as lightweight virtualization technology, has been widely deployed in cloud data centers and shows its great significance in performance improvement and cost reduction [16,21]. Many companies, such as Amazon and Google, have implemented their own container-based virtualization environment and provide an effective development pipeline for both service providers and end users. Compared with traditional VM technology, container guarantees better efficiency with fewer resources by sharing the host's OS system kernel using

namespace [4, 15] rather than requiring an OS per application. However, as a light-weighted version of VM, containers can only give an application level of abstraction, therefore fail to provide strong hardware isolation and complicated resource management. Note that the resource demands of different containers vary on both type and amount, i.e. FTP server as a typical bandwidth-sensitive application while Redis as a latency-sensitive one. When multiple containers run on one host machine and share the OS kernel with each other, resource competition can never be avoided and seriously hinders the system performance [10].

To tackle this issue, Docker uses CGroup [3] to enable CPU, memory, and disk resource allocation for containers. Sadly, network resource, required by thousands of containers within one host to perform intra and inter host communication, is totally ignored in present container framework. Therefore, it's critical to provide a network traffic control framework for container network.

To provide a lightweight network traffic control, Traffic Control (TC) [2] provided by Linux system is considered as an ideal candidate. It consists of three components: class, filter, and Qdisc. When a network packet from container arrives at TC module, it is firstly categorized into a class by the filter, and then line up at the Qdisc which belongs to the corresponding class. According to the network packet source and type, TC can create a large number of classes with different network resource allocation. These queueing packets will be further scheduled and sent to the destination according to the embedded control algorithms in Qdisc. Thanks to its lightweight and strong scalability, TC now is widely used in virtualized environment and scenarios for network traffic control. For example, Barker et al. [11] use TC to reduce service latency by balancing the tradeoff between sharing and dedicating network bandwidth for latency-sensitive applications running on VMs. While Ma et al. [17] propose a cluster management framework that leveraging TC to provide bandwidth limitation and network resource isolation for bandwidth-consuming applications.

Some researches also show the possibility and advantages of introducing TC into container systems [19]. Yet, we argue that TC cannot be directly applied to container network traffic control due to the following three factors. First, containers are running under multiple network modes to adapt to different workloads [12], e.g., overlay mode for allowing more flexibility and security in network management and non-overlay mode for achieving good performance but undermines security. TC, on the other hand, can only be functional in non-overlay mode which does not require packet encapsulation. While for containers in overlay mode, TC is invalid due to the failure of TC filters. TC has two filters: IP filter and Classid filter. IP filter fails in the packet encapsulation of overlay which will hide the container IP. Classid filter fails in the going across network namespaces of container packets because Classid is private information which only belongs to one namespace. That is, TC cannot provide a one-size-fits-all solution and fails to manage containers under multiple modes. Second, raw TC operation and configuration are relatively complicated and require a full understanding of the traffic control details. Through some tools like tcng [5] are proposed to simplify

the configuration. Yet they are designed for VM and cannot be applied to container management for the reason that tcng only simplifies the TC command into a script, the calculation process is still complicated and can only be operated by experts. Finally, TC only focuses on static network traffic control. That is, once a TC command is carried out, its organization and resource configuration cannot be changed. As a result, when a new container is activated, we are not able to adjust resource allocation of the existing containers with a global consideration of the resource requirements and quality of service. Similarly, if a launched container is deactivated, the freed resource cannot be reallocated to other containers to improve the resource utilization. Note that containers are short-lived and varied. In other words, each server can host hundreds or thousands of launched containers which are frequently activated and deactivated. The original TC resource allocation algorithm cannot deal with the high dynamic in the container environment and will lead to a performance degradation [12].

To overcome these three challenges, we propose a Container Network Traffic Control (CNTC) framework to ensure network resource isolation with careful consideration of container characteristics. Compared to TC, CNTC is a simple and easy-to-use container network traffic control framework, which provides the container with the bandwidth, priority, and latency control with a dynamic scheduling algorithm to achieve resource reallocation for higher utilization. The main contributions of this paper are summarized as follows:

1. We provide a detailed analysis of resource competition in container network. Through extensive experiments, we show that TC cannot even balance the network resource allocation between different containers. We then reveal the reason for the invalidity of TC under overlay mode by taking a close look at the design of TC.
2. Based on the above findings, we propose CNTC to provide network resource isolation and traffic control for containers by providing a container level resource scheduling strategies under overlay mode through Container Network ID (CNid). We further design a set of easy understanding APIs to simplify the configuration of the network resource in a what you see is what you get way. To deal with the dynamic of the container, a dynamic online resource allocation algorithm is also proposed to meet the variability of containers.
3. Finally, we implement a prototype of CNTC and provide extensive experiments on various use cases. The results show that the framework can guarantee bandwidth and latency constraints for the container in all network modes with relatively low overhead.

The rest of this paper is organized as follows: Section 2 explains the background and motivation, including network modes of the container and the architecture of TC which give an important guideline on designing CNTC. Then Section 3 describes the detailed design of CNTC. In Section 4, we evaluate the usability of the framework by using use cases. Section 5 discusses some related works and Section 6 concludes this work.

2 Background and Motivation

This section introduces network modes and characteristics of the container. We also analyze the design of TC, which helps us better understand the incompatibility between TC and the container. There are two reasons for the incompatibility between TC and the container as follows.

1) Incompatibility caused by Modes The design of container network modes is diverse falling into two categories: overlay and non-overlay. Overlay runs on top of another network to build virtual links between nodes. Many network modes leverage overlay, e.g., *Docker's native overlay network*. A container packet using overlay is encapsulated by VXLAN and host's UDP header as shown in Fig. 5 where we can see that the encapsulation hides the container IP. Another design is non-overlay, e.g., *NAT* which maps container IP to host to achieve connectivity. Overlay and non-overlay have their advantages and disadvantages to suit different workloads. The non-overlay doesn't need encapsulation, so it's simple but lacks security and flexibility. The overlay is the complement of non-overlay whose goal is to provide more security and flexibility for the network. It is noticeable that TC is invalid under the overlay. The reason is that TC has two filters to achieve packet identify, i.e. Classid filter and IP filter, which both fail to trace the container packet under multiple network modes. For Classid filter, Classid is one of the information of the packet which is private to one network namespace that means Classid filter is not designed for the container because a container packet needs to go across network namespace which leads to loss of Classid. As for the IP filter, it needs the container IP to trace packet. Sadly, we know that overlay hides the container IP which means IP filter only works well under non-overlay.

2) Incompatibility caused by Characteristics The container has the characteristics of short live, variability, and huge in numbers. For example, a survey of 8 million container usage reveals that 27% of containers have a lifetime no more than 5 minutes and 11% shorter than 1 minute [6] and Google Search launches about 7,000 containers every second [1]. However, TC is complicated and static which cannot meet the characteristics of containers. First, TC is implemented with various scheduling algorithms based on mathematical principles. If inexpert programmers want to achieve packet scheduling, they have to go through and obtain a deep understanding of these mathematical rules and traffic control details. Second, TC is static at its organization and resource configuration. When programmers try to use different restrictions on data streams, TC builds a hierarchical tree. Once the TC command is carried out, the tree cannot be changed. Moreover, the resource configuration of TC is also static. The TC algorithms only support to guarantee a static bandwidth, it doesn't support allocation by weight dynamically. Variability is one of the most important characteristics of containers. So when the number of containers decreases, the freed resource cannot be adjusted by TC to other containers for better resource utilization.

To validate the invalidity of TC on container network traffic control, we perform two experiments on both bandwidth and latency with two containers,

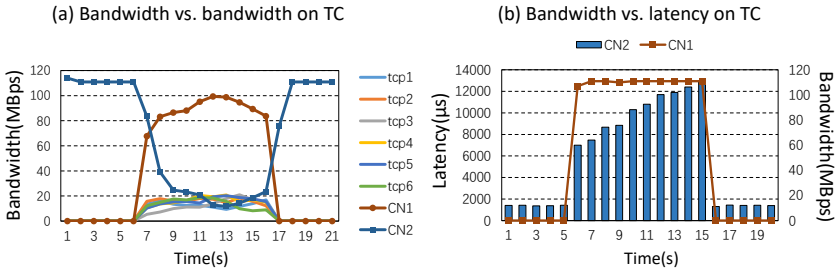

Fig. 1. Experiments on bandwidth and latency under overlay network using TC

i.e. CN1 and CN2, on the same host under *Docker's native overlay* mode. The benchmark is *Iperf*. CN1 is a bandwidth consuming container, which runs 6 TCP connections (tcp1-tcp6) in parallel and CN2 runs 1 TCP connection. Fig. 1(a) shows the result of bandwidth. From time slots 1-6, only CN2 is launched. From time slot 7, CN1 is activated and CN2 suffers a severe performance degradation. It can be observed that CN1 gets 6/7 of the total bandwidth while CN2 gets only the rest 1/7 because following the principle of link-fairness, the network resource will be allocated to the container according to the number of connections. Next, we evaluate the latency control of TC to guarantee a maximum 2ms latency. CN1 uses the same setting and runs an unrestricted TCP while CN2 runs *ping* under a guarantee. The results in Fig. 1(b) show that the activation of CN1 causes an increase in latency of CN2 and hinders the latency constraint.

The above two experiments reveal that TC is valid, complicated, and static. These incompatibilities also give us the guideline on designing a traffic control framework for the container network.

3 System Design

Through the above analysis and discussion, we conclude that the TC cannot meet the needs of the container network traffic control. To tackle this issue, we introduce CNTC, a container network traffic control framework shown in Fig. 2, targeting the following three goals.

1. **One size fits all:** The container has multiple network modes to suit different workloads, but container network traffic control is required in all cases. Therefore, CNTC should work well under all network modes.
2. **Easy to use:** The configuration of the TC is complicated. A series of easy-to-use APIs should be provided to enable container network traffic control with simple commands.
3. **Dynamic adjustment:** Containers are short-lived and varied. It means that CNTC must be able to automatically adjust the resources of containers when the status of containers changes.

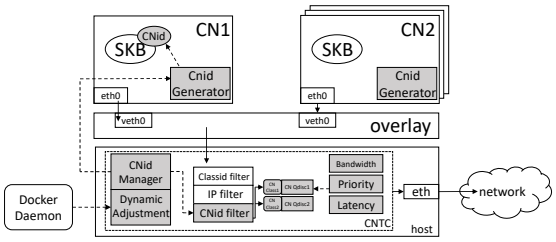

Fig. 2. Overview of CNTC

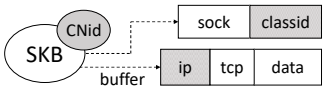

Fig. 3. Difference between CNid, Classid, and IP

3.1 CNid and CNid filter

As we mentioned above, TC is invalid under the overlay mode. To achieve a valid traffic control framework under different network modes for containers, we propose a new module as CNid and CNid filter to enable container-level packet tracing and resource allocation. To achieve this goal, two problems have to be solved: 1) How can we identify the source and destination container of each network packet after overlay encapsulation? 2) How to keep the identification of packets across network namespaces?

By analyzing the design of raw TC shown in Fig. 3, we can obtain that Classid and IP are only part of SKB (a structure keeping all control information of network packet) reference which will be lost when across network namespaces or will be encapsulated under overlay mode. So they cannot be used to trace and identify container packets. To solve this problem, we propose CNid to identify each container packet and a CNid Manager consisting of a bitmap to keep a record of CNid status, i.e. 0 for available and 1 for occupied. When a container is activated, CNid Manager looks for an available ID from the low bit of the bitmap. While if a container is deactivated, it can quickly release the CNid by setting its status as 0. Then, to ensure the traceability of the CNid, we also propose a CNid Generator which is in charge of inserting CNid into SKB structure as a member which leads to CNid not be encapsulated under overlay mode. By integrating CNid into the SKB structure, our CNid will not be released and re-created even when across network namespaces.

With the above design, Fig. 5 shows the procedure of a network packet from its source container to host. When a container is activated, the CNid Generator requests for an available CNid from CNid Manager. The Manager looks up for an available CNid and returns it to the Generator. Then a network stack call is made to sending a packet from the container namespace to the host namespace as shown in Fig. 4(a). The packet is sent via eth0 where veth_xmit is called in which the CNid Generator marks the CNid for SKB of the packet. Then, the packet goes across network namespace and joins the receive queue of the host CPU. Finally, It is received by the host network stack. After the packet arrives at overlay, Its SKB buffer will be encapsulated and sent the CNid filter for further process as shown in Fig. 4(b). CNid filter identifies the packet using

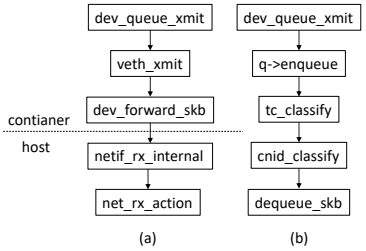

Fig. 4. Networking call stack

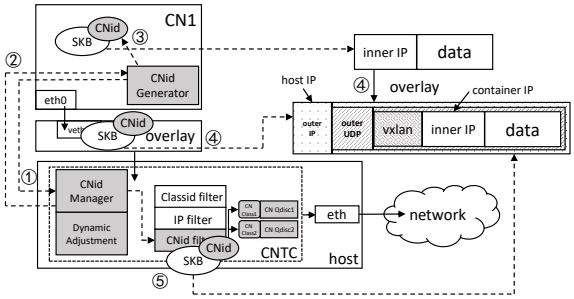

Fig. 5. Path of container packets

cnid_classify which gets the CNid from the SKB to generator the CN class ID. Then, cnid_classify uses the ID to find the CN class corresponding to the packet. Finally, the packet is added to the CN Qdisc of this class. When it's the turn of this packet to be sent, the packet is sent by dequeue_skb.

3.2 Simple APIs

TC is complicated, which has many traffic control algorithms falling into two categories: 1) Classless algorithms, which are used for constant types of data streams. 2) Classful algorithms, whose role is to use different restrictions on multiple types of data streams. We believe that classful algorithms are more suitable for the container which produce and process various types of data streams. However, to use classful algorithms, inexpert programmers need to understand the traffic control details and the mathematical formulas of the algorithms which are complicated just like what we mentioned in Section 2. Therefore, We propose these APIs in the form of container startup parameters shown in Table 1 to allocate network resources in the simple and easy style.

Take the latency as an example to show how we simplify the calculations. Generally, we use HFSC to guarantee latency which is one of the classful algorithms. To guarantee 100kbit bandwidth and 20ms latency guarantee for a container, the TC algorithm needs a command like *m1 250kbit d 8ms m2 100kbit* which has not a description of 20ms latency leading to confusing and complication. This command indicates a container has a normal bandwidth of 100kbit and during burst TC can send 250kbit in 8ms.

We simplify the command by a startup parameter in the style of container (*–rate 100kbit –latency 20ms*). We convert this parameter to TC command using the following two formulas. *8ms* can be calculated by Formula (1) where $Latency_I$ indicates the value of *–latency* and $Bandwidth_I$ indicates *–rate*. Note that following the principle of HFSC, when the latency of container C is guaranteed, other containers share this host are also allowed to send an MTU (Maximum Transmission Unit) size packet. That is to say, even the bandwidth-consuming container cannot occupy the network resource and some bandwidth needs to be reserved for the rest containers. Therefore, considering the reserved

Table 1. Startup parameters for container traffic control

Parameter	Description
rate	Static bandwidth for the container
weight	Dynamic bandwidth and dynamic priority for the container
priority	The order in which containers obtain idle bandwidth
latency	Maximum latency for the container

bandwidth, the actual process time of packets from container C as L_C can be calculated as

$$L_C = Latency_I - \frac{MTU}{Bandwidth_I} = 20ms - \frac{1500byte}{100kbit} = 8ms, \tag{1}$$

and the guaranteed bandwidth of container C as B_C should be

$$B_C = \frac{Bandwidth_I * Latency_I}{L_C} = \frac{100kbit * 20ms}{8ms} = 250kbit, \tag{2}$$

which provide the same latency and bandwidth guarantee as the raw TC commend.

3.3 Dynamic Adjustment

As we mentioned above, containers are short-lived and varied. Now, with the help of CNid, we enable the identification of container packets. Yet, how to deal with the online adjustment of traffic control and resource allocation is still an issue to be addressed. So we propose Dynamic Adjustment module to overcome the following two challenges: 1) How to adjust the TC organization? 2) How to adjust network resource?

Firstly, to adjust TC organization, Dynamic Adjustment needs to communicate with the Docker Daemon for monitoring the status of containers. When a container is activated, it requests an available CNid from the Manager and creates a new CN class and CN Qdisc. Secondly, Dynamic Adjustment should adjust the resources obtained by containers to achieve a higher utilization considering two factors: bandwidth and priority. Without loss of generality, we provide two adjustment ways taking into consideration of both rate and weight to allocate bandwidth for containers called rate-style and weight-style containers, respectively. Rate-style is a static allocation that guarantees the container to obtain a certain bandwidth while weight-style is a dynamic one which allocates bandwidth in proportion. Therefore, the static and dynamic containers are mixed on the same host. Changes in container status result in changes in rate and weight, ultimately affecting bandwidth allocation. In addition, weight also indicates dynamic priority, so changes in weight lead to changes in priority. Therefore, we propose a dynamic adjustment algorithm in algorithm 1 to reallocate the bandwidth and reconfigure priority.

Algorithm 1 DYNAMICADJUSTMENT

```
1: function Initial
2:     B_s ← 0, W_s ← 0, W_d ← 0
3:     B_d ← B_max
4:     {C_i} ← ∅
5: end function
6:
7: function ContainersActivationHandle(C_n)
8:     {C_i} ← {C_i} ⋃ C_n
9:     D{d_r, d_w, d_p, d_l} ← GetDemand(C_n)
10:    if d_w = NULL then
11:        B_d ← B_d − d_r
12:        B_s ← B_s + d_r
13:    else
14:        W_d ← W_d + d_w
15:    end if
16:    UpdateResource({C_i})
17:    PriorityHandle({C_i})
18: end function
19:
20: function UpdateResource({C_i})
21:    for {C_i} do
22:        D{d_r, d_w, d_p, d_l} ← GetDemand(C_i)
23:        W_s ← 0
24:        if C_i is rate-style then
25:            d_w ← (W_d / B_d) * d_r
26:            W_s ← W_s + d_w
27:        else
28:            d_r ← (B_d / W_d) * d_w
29:        end if
30:    end for
31: end function
```

Lines 1-5 are the initialization. The bandwidth is divided into two parts: static bandwidth and dynamic bandwidth which are represented by B_s and B_d. They are the bandwidth obtained by rate-style and weight-style containers. Similarly, the weight is also composed of two parts: W_s and W_d representing static weight and dynamic weight for rate-style and weight-style. $\{Ci\}$ indicates a network resource configuration for all containers.

The adjustment on bandwidth is handled by lines 10-15 and lines 20-31. When a static rate-style container is activated, static bandwidth increases while dynamic bandwidth decreases that means weight-style containers must reduce their bandwidth to satisfy the new rate-style container (lines 11-12). When a dynamic weight-style container is activated, dynamic weight increases while static bandwidth remains unchanged that means the new weight-style container does not affect the bandwidth of the rate-style containers, but the bandwidth of other weight-style containers is reduced due to the increase in dynamic weight leading to less bandwidth for each weight (line 14). After that, Dynamic Adjustment calls *UpdateResource* to update the resources of all containers. Rate-style container needs to update its weight because the change of weight sum leading to different weight per bandwidth (lines 25-26) and weight-style containers adjusts its rate according to the change of dynamic bandwidth (line 28).

To achieve adjustment on priority, *PriorityHandle* (line 17) is called. However, the priority way of TC is invalid for the container because TC has only eight priority levels. When containers' types of weight are more than eight, it

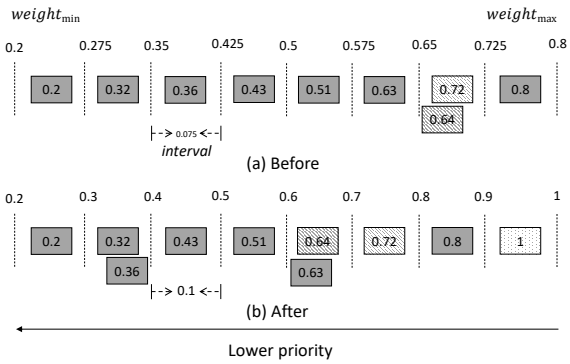

Fig. 6. Priority interval

is not possible to set priority for containers. So we propose a new way to set priority called *priority interval* where each interval represents a priority level which is shown in Fig. 6(a). When the type of weight is less than eight, Dynamic Adjustment set the priority in the positive order of weight. And if the type of weight is more than eight, the priority levels are not enough. Therefore Dynamic Adjustment leverages the *priority interval*. When the weight of a new activated container is out of range, the *priority interval* adjustment is triggered. Fig. 6(b) shows the *priority interval* after the activation of the container with weight 1 where we can see that the priorities of some containers have changed.

4 Evaluation

This section presents an evaluation of CNTC. The default experiment settings are listed as follows: 1) Hardware: experiments are performed on two servers connected by Gigabit Ethernet. Each server contains an Intel Core i5-7200 2.50 GHz CPU (2 cores) with 16 GB of DDR4 RAM and 256 GB HDD. 2) Software: we use Ubuntu 16.04.5 and Linux kernel 4.14.5. The Docker version is 17.09.0-ce. The overlay network is *Docker's native overlay* and the non-overlay network is *NAT*. 3) Benchmark: bandwidth benchmark is *Iperf* and latency benchmark is *ping*. We first evaluate the validity of the framework under the overlay network. Next, we observe the dynamic adjustment when the status of containers changes. Finally, we measure the overhead of the framework to see if it fits into a lightweight containerized environment.

4.1 Validity

Multiple network modes of the container are suitable for different application scenarios. But no matter what the scenario, it needs network traffic control. In this subsection, we evaluate the validity of CNTC under the overlay network of three aspects: bandwidth, latency, and priority.

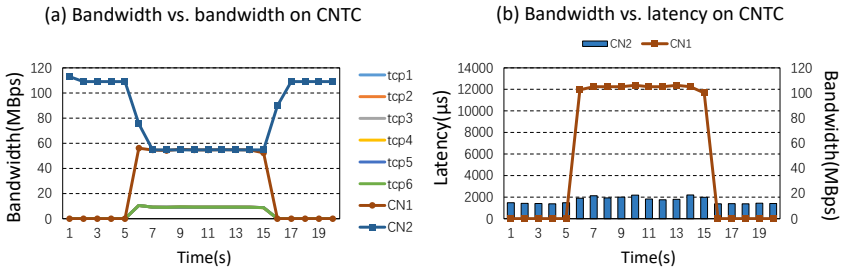

Fig. 7. Experiments on bandwidth and latency under overlay network using CNTC

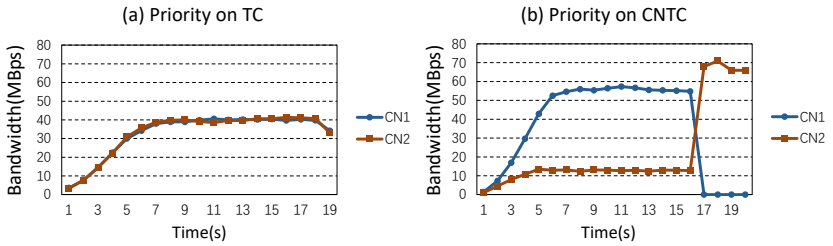

Fig. 8. Experiments on priority under overlay network using TC and CNTC

We repeat the experiments on section 2 to confirm the valid control of bandwidth and latency in our framework. Fig. 7(a) is the result of bandwidth which shows that when *container1* (CN1) is activated, the bandwidth of *container2* (CN2) drops, but it guarantees both containers get equal bandwidth which means CN1 cannot steal bandwidth by increasing the number of its connections. Next, Fig. 7(b) shows the latency experiment. When the unrestricted CN1 is activated, the latency of CN2 has a small fluctuation increase. But it hardly exceeds 2ms latency guarantee which means the framework meets the performance requirement set by CN2. Finally, we evaluate the valid control of the priority. The priority indicates the order in which containers obtain idle bandwidth. For better representing the competition for idle bandwidth, we set a minimum bandwidth for both containers leading to more idle bandwidth and set CN1 has a higher priority. Fig. 8(a) is the result on TC where two different priority containers are not treated differently. And Fig. 8(b) shows the result on CNTC where CN1 gets about 5 times the bandwidth of CN2, which embodies their priorities.

From the results of these experiments, we can conclude that CNid solves the problem of TC invalidity under overlay network mode.

4.2 Dynamic Adjustment

When the status of containers changes, CNTC should adjust the resources according to the change. We observe the dynamic adjustment of our framework by changing bandwidth and priority.

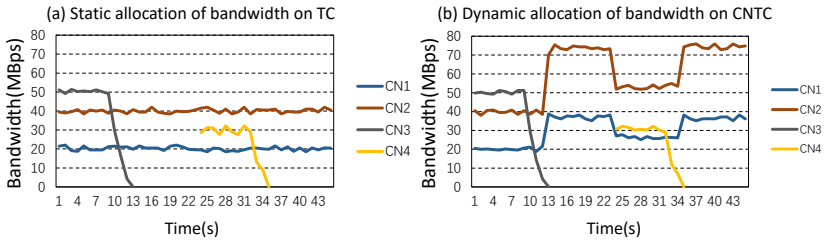

Fig. 9. Experiments on bandwidth adjustment

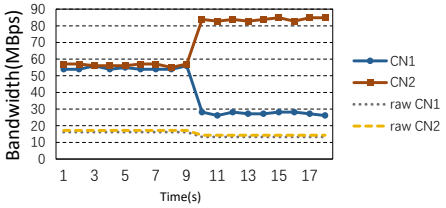

Fig. 10. Experiment on priority adjustment

First, we observe the adjustment of the bandwidth. We use 4 containers. *Container1* (CN1) and *container2* (CN2) are weight-style and their weight ratio is 1:2. *Container3* (CN3) and *container4* (CN4) are rate-style which are guaranteed 50MBps and 30MBps static bandwidth, and they are deactivated or activated during the experiment. We use TC and CNTC under *NAT* which is a type of non-overlay networks that make the TC valid. Since TC doesn't support allocation by weight, CN1 and CN2 can only be allocated in static by manual in TC experiment. For better observing the dynamic adjustment of bandwidth, we disable the idle bandwidth competition by priority which is shown in section 4.1. From Fig. 9(a), we can observe that CN1 and CN2 do not adjust the bandwidth allocation on TC which means the change of container status leads to a waste of bandwidth. Fig. 9(b) is the result of CNTC which achieves dynamic adjustment. When CN3 is deactivated at time slot 13, CN1 and CN2 obtain this 50MBps of idle bandwidth by a ratio of 1:2. And when the CN4 is activated at time slot 24, they shrink bandwidth to meet the requirement of CN4. Finally, they obtain this bandwidth again because of the deactivation of CN4. These experiments show that CNTC has higher utilization than TC.

Next, We evaluate the dynamic adjustment of priority. Take containers of Fig. 6(a) as an example. We select containers with weights of 0.64 and 0.72 (denoted as *container1* and *container2*) to observe change in priority. The other containers are still running on the host, but they do not send any data. Fig. 10 is the process of the experiment. Raw CN1 and raw CN2 are estimated by us which represent the bandwidth they should have obtained by weight. CN1 and CN2 represent the bandwidth actually obtained after their competition on idle bandwidth. Before time slot 10, *container1* and *container2* have the same

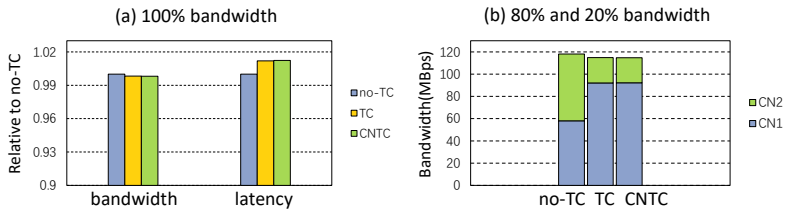

Fig. 11. Overhead of CNTC

priority, so they get equal idle bandwidth. At time slot 10, a container with a weight of 1 is activated (shown in Fig. 6(b)). We can observe two trends: 1) Raw CN1 and raw CN2 have dropped slightly because the increase of weight sum leads to less bandwidth for each weight. 2) The bandwidth actually obtained by containers has changed dramatically. The idle bandwidth obtained by *container2* is almost 5 times that of *container1* caused by new activated container which leads to the change of *priority interval* where *container2* becomes a priority over *container1*.

These experiments exemplify the great ability of multiple mode validity and dynamic adjustment of CNTC by the factor that it significantly improves the resource utilization and satisfies the service quality of all containers at the same time.

4.3 Overhead

The container is lightweight virtualization technology, so any traffic control framework with a high overhead is not appropriate. So in this section, we measure the overhead of CNTC.

First, we measure the case of allocating 100% bandwidth to a container to observe the overhead of bandwidth and latency under *NAT* using without resource control (no-TC), TC with IP filter, and CNTC. Fig. 11(a) shows the result which uses no-TC as the baseline, and the result of the other two cases are a relative value of the baseline. We can see that TC and CNTC have almost the same performance and a bandwidth degradation of only 0.16% relative to no-TC. The latency overhead is relatively obvious. The increases in latency of TC and CNTC are 1.19% and 1.24%, but their absolute value doesn't exceed 0.006ms. Therefore, in the absence of competition, our framework is almost the same as TC. And there is no obvious overhead relative to no-TC.

Next, we investigate the situation of competition. We use two containers that are allocated 80% and 20% bandwidth. Fig. 11(b) is the result. The bandwidth degradations of TC and CNTC are 2.7% and 2.8% which are slightly obvious than without competition. CNTC has a similar result to TC which reveals that the overhead is caused by packet scheduling instead of CNid.

It can be learned from the above experiments that our framework doesn't cause much overhead on the network, and it is suitable for the lightweight container network traffic control.

5 Related Work

Recently, many cloud resource management researches, especially network resource, have been proposed. Therefore, providing management for the network resource is important to both cloud providers and customers [8]. Current network resource management and traffic control researches can be divided into two categories: 1) third-party frameworks designed by the researcher and programmers, and 2) raw frameworks integrated into OSs and developing platforms like Linux TC.

1) Third-party Frameworks Shieh et al. [7, 18] consider the fairness of congested links in VM. However, link granularity control is too complex for hundreds of containers. In addition, they cannot guarantee a specific bandwidth which is used to estimate worst completion time that the applications really care about. Rodrigues et al. [9, 20] use a self-designed speed limiter for ingress and egress traffic. The limit of ingress needs congestion information between hosts, which makes the congested network worse. Popa et al. [13, 14] offer guaranteed allocations for network resources in switches which require hardware support. These approaches use a third-party framework in which they take into account too many factors that complicate the situation which leads to high overhead. These are not what the container wants since the container is designed for high performance.

2) Raw Frameworks TC is built into the Linux kernel and focuses on single host egress network traffic control. Many researches which require lightweight network traffic control leverage TC. Ma et al. [17] consider the container placement as a variable. Therefore, they achieve isolation and fairness through container scheduling. But they use TC to guarantee specific bandwidth to the container, which is only suitable for container clusters deployed under non-overlay networks. Herbein et al. [19] present mechanisms to allocate, throttle, and prioritize CPU, network bandwidth, and I/O bandwidth resources in containerized HPC environments. Their mechanisms use TC to achieve network traffic control. Barker et al. [11] use TC to manage the latency impact and to provide a choice between sharing and dedicated bandwidth for latency-sensitive applications. In summary, these works only use TC as a tool. They neither find the incompatibility between TC and the container, not customize the TC to better serve the container.

6 Conclusion

In this paper, we present CNTC, a framework to provide network resource isolation and traffic control for containers by leveraging lightweight network traffic control module TC. To simplify the network traffic control, we further provide a series of APIs to enable complicated network traffic control such as a specific bandwidth, priority, and maximum latency on each container. To tackle the dynamics of containers, an online resource management algorithm to adjust the resource allocation when the number and type of containers change. The evaluation results validate the efficiency and correctness of our CNTC framework.

References

1. Containers: The future of virtualization & sddc. https://goo.gl/Mb3yFq
2. Linux advanced routing & traffic control. https://www.lartc.org/
3. Linux control groups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
4. Linux namespace. https://www.kernel.org/doc/Documentation/namespaces/
5. Linux traffic control next generation. http://tcng.sourceforge.net/
6. The truth about docker container lifecycles. https://goo.gl/Wcj894
7. Alan, S., Srikanth, K., Albert, G., Changhoon, K., Bikas, S.: Sharing the data center network. In: Proceedings of NSDI. pp. 309–322. USENIX (2011)
8. C, M.J., Lucian, P.: What we talk about when we talk about cloud network performance. In: Proceedings of SIGCOMM. pp. 44–48. ACM (2012)
9. Henrique, R., Renato, S.J., Yoshio, T., Paolo, S., Dorgival, G.: Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In: Proceedings of WIOV. pp. 784–789. USENIX (2011)
10. Junaid, K., Eric, R., Wesley, F., Cong, X., et al.: Iron: Isolating network-based cpu in container environments. In: Proceedings of NSDI. pp. 1–17. USENIX (2018)
11. Kenneth, B.S., Prashant, S.: Empirical evaluation of latency-sensitive application performance in the cloud. In: Proceedings of ICMR. pp. 35–46. ACM (2010)
12. Kun, S., Yong, Z., Wei, C., Jia, R.: An analysis and empirical study of container networks. In: Proceedings of INFOCOM. pp. 1–9. IEEE (2018)
13. Lucian, P., Gautam, K., Mosharaf, C., Arvind, K., Sylvia, R., Ion, S.: Faircloud: sharing the network in cloud computing. In: Proceedings of SIGCOMM. pp. 187–198. ACM (2012)
14. Lucian, P., Praveen, Y., Sujata, B., Jeffrey, M., Yoshio, T., Renato, S.J.: Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. In: Proceedings of SIGCOMM. pp. 351–362. ACM (2013)
15. Prateek, S., Lucas, C., Prashant, S., YC, T.: Containers and virtual machines at scale: A comparative study. In: Proceedings of Middleware. pp. 1–13. ACM (2016)
16. Rajdeep, D., Reddy, R.A., Dharmesh, K.: Virtualization vs containerization to support paas. In: Proceedings of IC2E. pp. 610–614. IEEE (2014)
17. Shiyao, M., Jingjie, J., Bo, L., Baochun, L.: Maximizing container-based network isolation in parallel computing clusters. In: Proceedings of ICNP. pp. 1–10. IEEE (2016)
18. Sivasankar, R., Rong, P., Amin, V., George, V.: Netshare and stochastic netshare: predictable bandwidth allocation for data centers. In: Proceedings of SIGCOMM. pp. 5–11. ACM (2012)
19. Stephen, H., Ayush, D., Aaron, L., Sean, M., Jose, M., Yang, Y., Seetharami, S., Michela, T.: Resource management for running hpc applications in container clouds. In: Proceedings of ISC High Performance. pp. 261–278. Springer (2016)
20. Vimalkumar, J., Mohammad, A., David, M., Balaji, P., Changhoon, K., Albert, G.: Eyeq: Practical network performance isolation at the edge. In: Proceedings of NSDI. pp. 297–312. USENIX (2013)
21. Wes, F., Alexandre, F., Ram, R., Juan, R.: An updated performance comparison of virtual machines and linux containers. In: Proceedings of ISPASS. pp. 171–172. IEEE (2015)