Network Quality of Service in Docker Containers

Ayush Dusia
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
Email: adusia@udel.edu

Yang Yang
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
Email: yyangwin@udel.edu

Michela Taufer
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
Email: taufer@udel.edu

***Abstract*—This poster presents an extension to the currently limited Docker's networks. Specifically, to guarantee quality of service (QoS) on the network, our extension allows users to assign priorities to Docker's containers and configures the network to service these containers based on their assigned priority. Providing QoS not only improves the user experience but also reduces the operation cost by allowing for the efficient use of resources. Our implementation ensures that time-sensitive and critical applications, hosted in high-priority containers, get a greater share of network bandwidth, without starving other containers.**

I. Introduction

In order to ensure high-quality performance for critical applications executed in Docker's containers, a required level of service should be ensured without expanding or overprovisioning the network. Unfortunately, Docker's networks are currently configured to provide the "best effort" to all the traffic; and parameters such as bandwidth, reliability, and packets per second for a specific application cannot be guaranteed. Consequently a single bandwidth-intensive application results in poor or unacceptable performance for any other application sharing the Docker network. This problem can be solved by introducing quality of service (QoS) mechanisms that provide preferential treatment to traffic and applications. Since the Docker networking is in its infancy (i.e., it offers limited options to configure network usage), it does not provide QoS yet. In this poster, we address this problem by proposing a QoS mechanism that enables preferential delivery service for critical applications in Docker, while ensuring sufficient bandwidth, controlling latency and delay, and reducing data loss.

A virtual Ethernet bridge, *docker0*, is created when Docker boots up. By default, all the containers are configured to be in the same subnet and to use *docker0* so that they can communicate with one another. For each container, a pair of virtual Ethernet interfaces is created, and an IP address is assigned to the container. Currently, no options are available to configure network shares, bandwidth, and priority, as in the case of other resources such as CPU and memory. Thus each container gets an equal share of the bandwidth. Assuming that each container is dedicated to host only a single application, a container for a real-time application requires higher priority over regular applications hosted on other containers. Thus, Docker needs QoS mechanisms that provide differential services to containers with respect to network bandwidth and that are based on some priority criteria, as our proposed mechanism does.

Figure 1 shows the architecture of our implementation in Docker. The implementation consists of a packet classifier and priority scheduler. The packets in the flows are classified and added to one of the three available priority queues. The scheduler dequeues the packets and sends each packet to a container according to the queue's priority. Our implementation provides the functionality to assign priorities to containers. The priority values are high, medium, and low, where medium is the default value that is assigned to a container. The packet classifier and scheduler are built on top of the *docker0* bridge, which prioritizes the network access of containers. A higher share of the total available network bandwidth is provided to the containers with higher priority.

We tested our scheduler with Docker version 1.5, which is installed on Ubuntu 14.04 LTS. The results of our experiments show that our scheduler implementation efficiently provides QoS to containers based on their priorities.

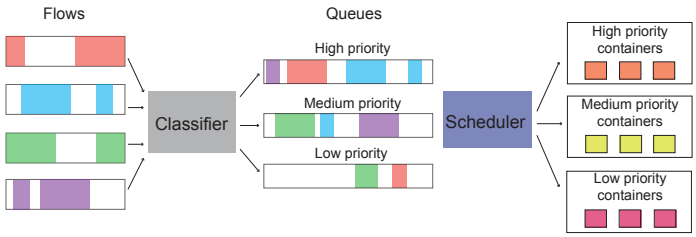

Fig. 1. Architecture of our priority queue scheduler.

II. Methodology

We use the Linux traffic control (TC) utility [1] to configure the kernel to shape, schedule, and classify network traffic in and out of an interface. The TC utility configures a queuing discipline, referred to as *qdisc*, for the ingress and egress directions. When the applications send out packets, the packets are enqueued to the configured egress *qdisc*. Similarly, the packets that are received by the system are enqueued to the configured ingress qdisc. Thus, the packets from the *qdisc* either are sent to the network adapter driver to send them out of the system or are sent to the applications running on the system that request the packets. The default *qdisc* configured on all interfaces is "Packet limited First In, First Out queue" (pfifo), which is the simplest queue with no processing and therefore has no overhead. The *qdisc* is either classful or classless. The classful *qdiscs* are useful to provide differential service to different flows of traffic by adding classes and filters

to *qdiscs*. Each class can also be configured to have a *qdisc* with subclasses. The classless *qdiscs* have no subclasses; they do basic management of traffic by reordering, delaying, or dropping packets.

We implemented the PRIO scheduler *qdisc* on the *docker0* virtual Ethernet bridge of Docker. The PRIO *qdisc* is a classful queuing discipline that contains an arbitrary number of classes with priorities. We configured the *qdisc* to include three classes and then added the Stochastic Fairness Queueing (SFQ) *qdisc* to each class. An SFQ does not shape traffic but only schedules the transmission of packets, based on "flows." The goal of SFQ is to ensure fairness: each flow is able to send data in turn, thus preventing any single flow from drowning out the rest. The classes are dequeued in numerical ascending order of priority. The PRIO *qdisc* acts as a scheduler and does not delay packets; it is also useful for lowering latency when traffic does not need to be slowed. Each class acts as a priority queue, where class 1 has the highest priority and class 3 has the lowest priority. We use a filter based on the IP address to determine the class to which a packet will be queued. A priority is assigned to the containers when they are created. Since each container has an unique IP address, a filter rule for that IP address is added to the PRIO *qdisc*. All packets are checked for destination IP address and then enqueued to a class based on the IP filter rule of the PRIO *qdisc*. For example, all the packets that have destination IP address of the containers with high priority are enqueued to class 1. Similarly, the packets with destination IP address of the containers with default and low priorities are added to class 2 and 3, respectively. The packets are enqueued to a class as they arrive, but they get dequeued in the ascending order of the class. The PRIO *qdisc* scheduler checks for packets in the queue of class 1; if no packets are available to dequeue, the queue of class 2 is checked, and then similarly the queue of class 3 is checked. The dequeuing of packets from the queue of different classes enforces the scheduling policy and priorities to containers.

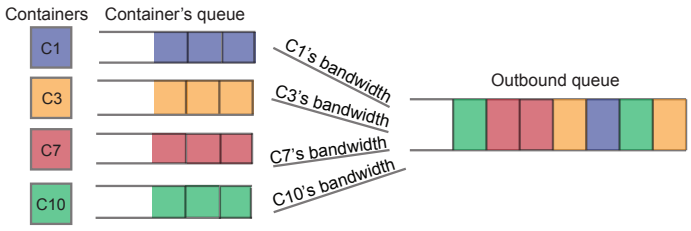

Fig. 2. Outbound network traffic configuration for containers.

We also added a functionality to throttle the rate at which the packets are sent out by a container. Figure 2 shows that an outbound bandwidth is configured for each container. The bandwidth is assigned to the container's *veth* interface in the container's namespace. This rate is set by the user when the container is created. The rate is enforced by using the token bucket filter (TBF) classless queuing discipline to shape the network traffic by configuring rate, burst size, and limit size parameters of the *qdisc*. Rate is a user-defined throttle speed. Configuring a TBF *qdisc* for a particular rate usually has an overhead of 40% to 50% of the configured rate. We reduce this overhead to 10% by configuring the *qdisc* parameters for optimal performance—specifically, by setting the burst size to the 10% of the user-defined throttle speed.

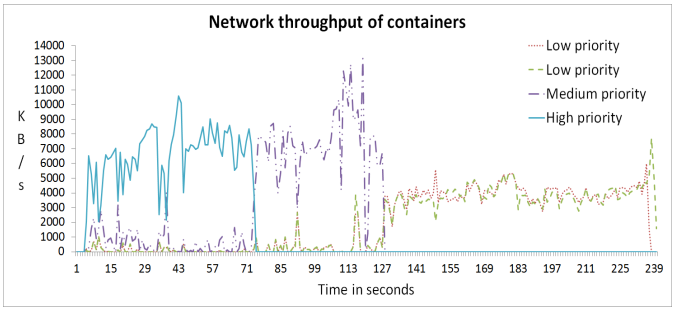

Fig. 3. Throughputs of low-, medium-, and high-priority containers.

III. Test Setup and Results

We set up an FTP server to host files of size 450 MB, and we created containers that execute a script to download these files. Specifically, for this experiment we created a container with high priority, a container with default (i.e., medium) priority, and two containers with low priority. The script to download the files using FTP was started simultaneously in all four containers. We then monitored the network throughput of each container using a utility called Nethogs. We also compared network throughput using Wireshark.

Figure 3 shows the results of the experiment. We observe that the container with high priority has the highest share of the total throughput until the file download completes. Then the container with the default (i.e., medium) priority gets the highest share of the total throughput. After the file download completes in the medium priority container, the containers with low priority gets all the bandwidth. The results also show that the low-priority containers get an equal share of the total throughput. These results prove that our scheduler implements QoS based on the priority of the containers and that the containers with equal priority get an equal share of the available bandwidth.

IV. Conclusion

Our extension to the Docker networking presented here guarantees QoS to containers, so that their network bandwidth matches assigned priority. Providing QoS to containers has two advantages: (1) containers hosting user-sensitive applications, such as real-time multimedia, or some high-bandwidth applications can now be assigned higher priority; and (2) operating costs can be reduced by using existing network resources more efficiently and thus delaying or reducing the need for expansion or upgrades. Moreover, when containers host applications using UDP, which is not sensitive to network congestion, our QoS implementation allows such containers to be throttled appropriately to achieve the desired levels of bandwidth sharing across all containers. The complete source code of our implementation is available on GitHub at [2].

References

[1] Linux Advanced Traffic Control. `http://lartc.org/howto/`.

[2] Codebase. `https://github.com/adusia/docker`.