Design and Implementation of Ceph Block Device in Userspace for Container Scenarios

Li Wang
College of Computer, NUDT
Changsha, China

Yunchuan Wen
Kylin Corporation
Changsha, China

***Abstract*—Ceph is a well-known and widely deployed open source distributed storage. Specifically, it is the mostly used storage backend for popular OpenStack cloud computing platform. For the traditional usage of Ceph in cloud computing, Ceph block device implemented in the VMM (virtual machine monitor), *qem-rbd*, is used to provide disks for the VMs (virtual machine). Recently, the container technology becomes more and more popular, and is regarded as a promising alternative solution to virtualization for cloud computing. A container is much more lightweight than a VM, thus achieves better performance. The technology behind container is different with virtualization, and VMM is not used, so *qem-rbd* could not be used in container scenarios. Currently, the only choice to integrate Ceph with container to provide disks to containers is to use the Ceph block device implemented in OS (operating system) kernel, *ker-rbd*. However, *ker-rbd* has some important limitations. In this paper, we propose the design and implementation of *use-rbd*, Ceph block device implemented in userspace. *use-rbd* has a lightweight yet efficient design, which makes it easy to be implemented correctly and stably. In contrast to *ker-rbd*, *use-rbd* has better portability and maintainability. Our experimental results show that *use-rbd* enjoys good performance scalability, and delivers the same or sometimes slightly higher performance than *ker-rbd*, therefore is a promising solution for container scenarios. *use-rbd* has been accepted and merged into Ceph official code, and freely available for downloading and using.**

***Keywords*-Container; NBD; Userspace; Block Device;**

I. Introduction

Cloud computing is regarded as the next generation computing infrastructure. The companies could benefit from cloud computing by improved resource utilization, reduced cost and easier system maintenance. OpenStack [1] is one of the largest and most active open source cloud computing platforms, and according to the recent statistics published in OpenStack Summit 2015 in Tokyo, 62% of all Openstack deployments are using Ceph block device (*qem-rbd*) as the storage for VM images and instances.

Ceph [2], [3] is a famous open source distributed storage. A Ceph storage cluster consists of two types of nodes: MON and OSD. MON is responsible for monitoring the health state of the cluster, maintaining a global and authoritative view of the cluster, and synchronizing the state in the cluster nodes. OSD is responsible for data storage and management, and it handles the data read and write operations from the clients.

Recently, container receives more attention and is considered as an promising alternative technology to traditional virtualization for cloud computing. Container is a technology which allows an OS kernel to run multiple isolated userspace instances (containers), each container feels like a OS from the point of view of its users and applications. In contrast to virtualization, which relies on VMM running on the host OS to present a complete set of hardware - CPU, memory, disk, etc - to the guest OS, container focuses on process and resource isolation and containment rather than emulating a complete physical machine. For the container scenarios, each container shares the host OS kernel with other containers. The containers run directly in the host OS, and use the host OS's normal system call interfaces, and do not need to be run on top of a VMM. That means the containers are much more lightweight and use far fewer resources than VMs. The other advantages of container over virtualization include simpler deployment and faster boot, etc. The disadvantage is that each container must use the same OS as the host OS.

For the container scenarios, there does not exist VMM, as explained in Section II, *qem-rbd* could not be used. To use Ceph as the storage backend for containers, currently the only choice is to use *ker-rbd*, which is a block device driver implemented in the OS kernel. However, as detailed in Section II, there are some drawbacks for using *ker-rbd* in the container scenarios. In this paper, we make the following contributions,

- We introduce the design and implementation of *use-rbd*, Ceph block device in userspace. *use-rbd* runs in the userspace, with good portability and maintainability.
- Our evaluation shows that *use-rbd* demonstrates good performance scalability, achieves the same performance as *ker-rbd*, and sometimes even outperforms it.

The rest of this paper is organized as follows. For background information, Section II describes the deficiencies of *ker-rbd* and motivates the development of *use-rbd*. Section III presents the design and implementation of *use-rbd*. Section IV evaluates *use-rbd*. Section V discusses related

978-1-5090-3071-2/16 $31.00 © 2016 IEEE
DOI 10.1109/IS3C.2016.105

work. Section VI concludes the paper.

II. Motivation

In the virtualization scenarios, as shown in Figure 1, the *qem-rbd* is implemented as a block device driver in QEMU, the VMM to provide disks for the guest OS in the VM. When the guest OS accesses the disk, QEMU will intercept the requests and pass the control to *qem-rbd*, which will invoke *librbd* to communicate with Ceph cluster to serve the requests. *librbd* is a library which encapsulates the details and provides the caller a set of simple and generalized interfaces to interact with Ceph cluster.

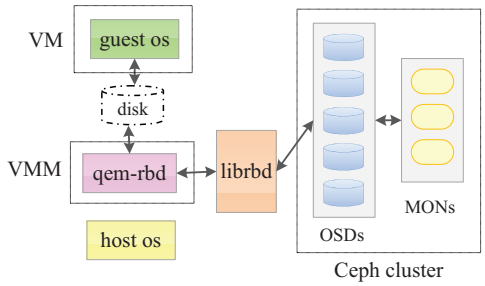

Figure 1. The architecture of *qem-rbd* for virtualization scenarios.

As described above, *qem-rbd* is implemented in the VMM to provide disks to guest OS in the VM. For the container scenarios, they do not use virtualization, thus VMM is not used, so *qem-rbd* could not be used in this case. Containers run directly in host OS, so the Ceph block device should provide disks in host OS. Currently, *ker-rbd* is the only choice. *ker-rbd* is implemented as a block device driver inside the Linux kernel as shown in Figure 2. However, there are the following drawbacks for *ker-rbd* to be used in container scenarios,

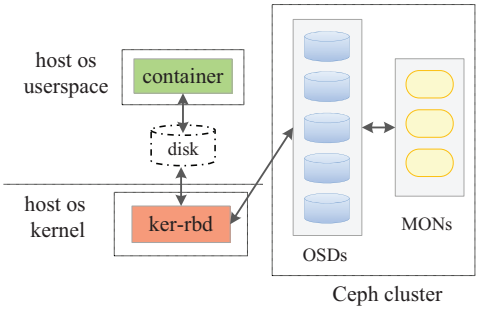

Figure 2. The architecture of *ker-rbd* for container scenarios.

(1) Since *ker-rbd* is implemented inside the kernel, it could not call the *librbd* which runs in the userspace. As a result, *ker-rbd* has to re-implement the functionality of *librbd* in the kernel. Driven by the wide deployment and heavy use of *qem-rbd*, *librbd* is under rapid development and improvement, while the development of *ker-rbd* is behind that of userspace *librbd*, without some useful features;

(2) Due to the kernel implementation nature, the flexibility and portability (dependent on kernel version) of *ker-rbd* are not very good;

(3) More importantly, according to our experience on ARM64 architecture for container scenarios, there exist stability issue for *ker-rbd*, easily cause kernel hang or disk utilization 100%.

These motivate us to develop Ceph block device in userspace, being able to provide disks in host OS, and could rely on *librbd* to interact with Ceph cluster rather than re-implementing the functionality. The userspace implementation nature also makes it have better flexibility, portability and maintainability, also its problem will not make the system unusable.

III. Design and Implementation

Linux provides NBD (network block device) infrastructure which works inside the Linux kernel, and applications could create disks (block devices) through it. Essentially, NBD just implements the application interact interfaces, but it does not take care of the data management, instead, whenever the applications try to access the NBD disks, it intercepts the requests by NBD disk driver inside the kernel, and forwards them into user space through a network socket.

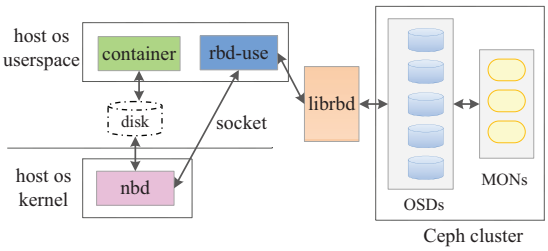

Figure 3. The architecture of *use-rbd* for container scenarios.

NBD infrastructure enables us to implement Ceph block device in user space, call it *use-rbd*. The architecture is shown in Figure 3. *use-rbd* listens on the NBD socket for incoming application disk access requests forwarded by NBD kernel driver, and when it receives the requests, it calls *librbd* to communicate with Ceph cluster through network connections, and relays the requests to Ceph OSDs to asynchronously process the requests. *use-rbd* registers the call back interfaces of *librbd*. When *librbd* receives responses from Ceph OSDs, it will invoke the call back interfaces. Then *use-rbd* gets the responses and relays them to NBD kernel driver through the NBD socket. NBD kernel driver parses the responses and returns results to the applications. From the point of view of the applications, they feel like the data are from local disks, and do not know the existence of Ceph cluster.

Algorithm 1 The process by reader_thread.

```
1: procedure reader_thread_entry
2: while stop == FALSE do
3:     request = wait_for_request(socket);
4:     if request→command == NBD_CMD_DISC then
5:         return
6:     end if
7:     if request→command == NBD_CMD_WRITE then
8:         recv_the_data(socket, request→data, request→data_len)
9:     end if
10:    register_callback(librbd_thread_entry, request→call_back, request)
11:    lock(thread_lock)
12:    add_to_list(request, pending-reqeusts)
13:    unlock(thread_lock)
14:    if request→command == NBD_CMD_READ then
15:        librbd→aio_read(request→from, reqeust→data,
16:          request→data_len, request→call_back)
17:    end if
18:    if request→command == NBD_CMD_WRITE then
19:        librbd→aio_write(request→from, request→data,
20:          request→data_len, request→call_back)
21:    end if
22:    if request→command == NBD_CMD_FLUSH then
23:        librbd→aio_flush(request→call_back)
24:    end if
25:    if request→command == NBD_CMD_TRIM then
26:        librbd→aio_discard(request→call_back)
27:    end if
28: end while
29: end procedure
```

use-rbd adopts a thread *reader_thread* for processing the requests as described in Algorithm 1. In Line 3, it listens on the NBD socket, whenever receives a request, in Line 4, it checks if it is a disconnect request, if so, it returns. Otherwise, in Line 7, if it is a data write request, *reader_thread* further receives the data to be written from the socket. Next, it registers the call back function of the *librbd*, then it inserts the request into a list *pending-reqeusts*. In Lines 14 – 27, it invokes the corresponding *librbd* method according to the operation type of the request, to process the request. The *librbd* methods are all performed asynchronously, so *reader_thread* does not wait for the finish of the process, and returns. The design makes *reader_thread* could focus on receiving the requests, increase the speed of request response.

Algorithm 2 The process by librbd thread.

```
1: procedure librbd_thread_entry
2: Input: ret, request
3: if ret < 0 then
4:     request→reply.err = ret
5: else
6:     if request→command == NBD_CMD_READ or
         request→command == NBD_CMD_WRITE then
7:         if ret != request→data_len then
8:             request→reply.err = -EIO
9:         end if
10:    end if
11: end if
12: lock(thread_lock)
13: remove_from_list(request, pending-requests)
14: add_to_list(request, finished-reqeusts)
15: wake_up(thread_lock)
16: unlock(thread_lock)
17: end procedure
```

Algorithm 3 The process by writer_thread.

```
1: procedure writer_thread_entry
2: while stop == FALSE do
3:     lock(thread_lock)
4:     while list_is_empty(finished-requests) and stop == FALSE do
5:         sleep_on_lock(thread_lock)
6:     end while
7:     if list_is_empty(finished-reqeusts) then
8:         unlock(thread_lock)
9:         return
10:    end if
11:    request = get_front(finished-requests)
12:    remove_from_list(reqeust, finished-requests)
13:    unlock(thread_lock)
14:    send_the_reply(request→reply, socket)
15:    if request→command == NBD_CMD_READ and
         request→reply.err == 0 then
16:        send_the_data(socket, request→data, request→data_len)
17:    end if
18: end while
19: end procedure
```

librbd maintains a thread pool to improve parallelism. Multiple requests could be handled simultaneously by different *librbd* threads, and finished out-of-order. When the *librbd* thread receives the response from Ceph OSDs, it will invokes the call back function registered by *reader_thread*, which is *librbd_thread_entry*, as shown in Algorithm 2. In Lines 3 – 11, the *librbd* thread checks the result of the process, and sets the error code if necessary. Then it removes the corresponding item from *pending-requests* list and insert it as well as the response into *finished-requests* list, and sends a signal to wake up the *writer_thread*. Here we introduce a dedicated thread *writer_thread* to be responsible for sending the responses back to NBD rather than leaving the *librbd* thread do it, since the NBD network socket must be communicated in an synchronous and atomic way, multiple threads are not allowed to write the socket simultaneously. Although a lock can be introduced to avoid the competition, it will waste time for *librbd* threads to compete the lock, and the threads have to wait if not succeed.

The Algorithm for *writer_thread* is shown in Algorithm 3. In Lines 4 – 6, when the *finished-requests* list is empty, it sleeps. When it is waked up, it will get the response from *finished-requests* list, and send it to NBD through the socket in Lines 11 – 14. In Lines 15 – 17, if the request is a data read request, it further sends the data read to NBD.

IV. Evaluation

The test bed is six workstations, with each being an Intel(R) Core(TM) i7-4770 CPU at 3.40GHz with 8 GB memory. The operating system is Ubuntu 14.04 x86 64 server with the Linux kernel 3.19. The Ceph version is 10.0.0.2. The Ceph cluster comprises 1 MDS, 1 MON and 4 OSDs. The number of data copies is two.

We use *fio* to measure the data read and write performance, with the parameters being '-direct=1 -iodepth=16 -ioengine=libaio -bs=512K'. Figure 4, Figure 5, Figure 6 and Figure 7 show the performance measured on *use-rbd* relative

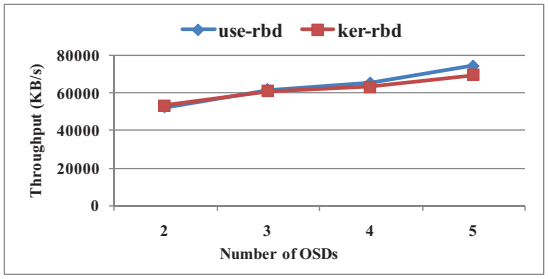

Figure 4. Throughput of *use-rbd* over *ker-rbd* for sequential read under different number of OSDs.

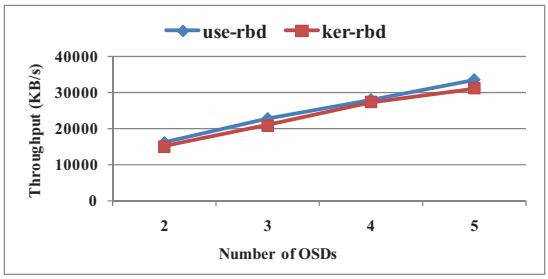

Figure 5. Throughput of *use-rbd* over *ker-rbd* for random read under different number of OSDs.

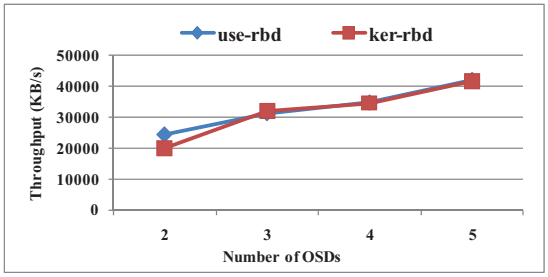

Figure 6. Throughput of *use-rbd* over *ker-rbd* for sequential write under different number of OSDs.

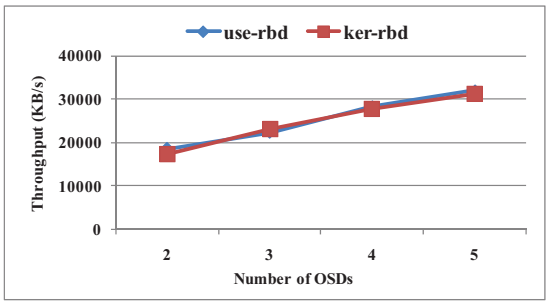

Figure 7. Throughput of *use-rbd* over *ker-rbd* for random write under different number of OSDs.

to *ker-rbd*, under the patterns of sequential read, random read, sequential write and random write, respectively. The evaluation are done under different number of Ceph OSDs. As the results shown, *use-rbd* has good performance scalability, and for all types of data read and write operations, *use-rbd* achieves the same or even a little bit higher performance compared to *ker-rbd*.

V. Related Work

Sheepdog [4] is a distributed storage, like *qem-rbd*, it implements a block device driver in QEMU to provide disks to guest OS. GlusterFS [5] is a distributed file system which is also integrated with QEMU to be able to provide disks to guest OS. While NBD [6] enables to implement block device in userspace, FUSE [7] provides an infrastructure to implement a file system in userspace.

VI. Conclusion

We have presented a Ceph block device implemented in userspace, *use-rbd*, for container scenarios. *use-rbd* communicates with Ceph distributed storage through *librbd* to serve data requests to respect its flexibility, stability and reliability. Based on the support by NBD infrastructure, *use-rbd* is able to receive the disk access requests relayed from kernel, and send the responses back into the kernel. The lightweight yet efficient design makes it easy to be implemented correctly and stably. Our experimental results demonstrate that *use-rbd* achieves almost the same or even slightly higher performance than *ker-rbd*, the Ceph block device implemented in kernel. Thus *use-rbd* is very practical, and is a promising solution for container scenarios.

Acknowledgment

We wish to thank Mingxin Liu and Xianxia Xiao for the help on experiments and writing. This research is supported by the National Natural Science Foundation of China (61370018).

References

[1] "Openstack project," *www.openstack.org*.

[2] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320.

[3] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "Rados: A scalable, reliable storage service for petabyte-scale storage clusters," in *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, ser. PDSW '07. New York, NY, USA: ACM, 2007, pp. 35–44.

[4] K. Morita, "Sheepdog: Distributed storage system for qemu," in *KVM Forum 2010*, 2010.

[5] V. Bellur, "Integrating glusterfs, qemu and ovirt," in *Linux Conference Europe 2013*, 2013.

[6] "Nbd project," *http://sourceforge.net/projects/nbd*.

[7] "Fuse project," *http://sourceforge.net/projects/fuse*.