Inter-VM communication library using IVSHMEM

Pratik Fegade, 120050004
Devdeep Ray, 120050007
Department of Computer Science
IIT-Bombay

November 24, 2015

Abstract

Communication between co-located VMs is required often. Some web services their web server and their database server on different VMs, but often on the same host. Virtual NIC communication is slow because of multiple copy operations that take place, and this report aims to explore a method which does zero copy communication between two VMs on the same host and looks at how this performs compared to the default socket communication that qemu-KVM provides.

1 Introduction

Most data centers resort to virtualization to run processes because virtualization has many advantages, like the ability to migrate to another machine without stopping the service for too long, guarantee resource availability, etc. Certain services run different sub services on the same host, but in different VMs running on the same host. These sub services often communicate with each other via sockets. This results in a performance impact, because of multiple copy operations done on the packet. First, it gets copied from the process space to kernel space of the VM. It then gets copied to the virtual NIC, which will be some memory in the VMM's memory space. Then

it is copied to the kernel space of the other VM, and finally it reaches the user space of the other process. Hence, there are 4 copies that happen for each packet. Qemu provides a virtual PCI device called IVSHMEM. This is backed by a shared memory region in the host. We try to use this shared device to communicate between VMs. We write a library and a daemon that runs on the host to give access to this device to the processes, and analyze performance of this system as compared to simple socket based communication. The library also provides synchronization mechanisms so that the processes have a Linux shared memory like API to talk to each other and to coordinate access to the shared memory. The host daemon also arbitrates access to sections of the shared memory for providing multiple shared memory chunks using a single IVSHMEM device.

2 Previous work

DPDK is a solution which attempts to use IVSHMEM for inter-VM communication. It is a polling based library and does not run a daemon on the host. It uses the shared device for all its communication and synchronization. It's interface is a packet based networking interface. There is a IVSHMEM system called NAHANNI, which is a barebones system. It just provides a driver for the PCI device, mapping it to a file in the guest. In our tests, we could not get the NAHANNI driver to work. Instead, we used a driver written by Chinmay Kulkarni for the IVSHMEM PCI device. The driver mounts the device as a file in the guest file system.

3 Architecture overview

3.1 Host

On starting a qemu guest with IVSHMEM enabled, a shared memory region gets created on the host. This memory will be shared among all the guests as well. We have a daemon that runs on the host. This is written in python. This daemon keeps track of the locking system and memory allocation to each process. The guest library communicates with this daemon whenever it wants to modify the condition variables, and also to request for shared memory blocks.

3.2 The daemon

The daemon provides socket based RPC for different types of communication from the guest. It supports allocation of shared block, deallocation of shared block, creation of condition variable, locking a condition variable, signalling and waiting on a condition variable. It does this by maintaining information about which process acquired a condition variable, which processes are waiting, which processes are waiting to acquire a condition variable, which process is allocated which block of shared memory, etc. It would be good to have a heart beat system so that locks held by dead processes are released after a timeout, but we have not implemented that. The daemon based system prevents busy waits in the guests.

3.3 Memory partitioning

The host daemon allocates memory to the guests out of the entire shared memory pool. It allocates memory in blocks, and the size of the block can be configured in the host before starting the daemon. It maintains read and write permission information for each block. We also have a write exclusive mode, which ensures no other process can get permission to write to that block. We maintain a free list of blocks and we pick from this to allocate to processes. Hence, allocation of shared memory region is constant time.

3.4 Guest

In the guest, the library is in C++. It provides userspace functions to allocate, deallocate, lock, release, wait and signal. Initially, it mprotects all the shared memory region, so that the process cannot do illegal access to the memory. It also manages the return codes from the host daemon. On allocation, it converts the response from the daemon to an address, and unprotects that range of memory if everything was alright. On release of a memory block, the library protects that region of memory again.

4 Example application using the library

In Fig. 1 and Fig. 2, we give sample code an as example of the usage of the library. Fig. 1 is a code for a writer that writes onto a shared memory area after making sure that the area is empty (i.e. the data written previously

has been already read by the reader). Fig. 2 is the corresponding reader. They use the first byte of the shared memory area to store the information whether the area is empty or not in a bool. Both the reader and the writer then check this bool by taking a lock, and then read or write respectively if the shared area is full or empty.

```
while(i < to_transfer) {
  // Check shared variable
  shm_acquire_cv(cvid, uid);
  if (!(*((bool*)offset))) {
    // Data not available, write here
      memcpy((void*)((bool*)offset + 1),
        data, size - sizeof(bool));
    ++i;
    *((bool*)offset) = true;

  // Notify the reader
    shm_notify_cv(cvid, uid);
  } else {
    // Wait for the reader
    shm_wait_cv(cvid, uid);
  }
  shm_release_cv(cvid, uid);
}
```

Figure 1: A Bare Bones Writer Loop

5 Results

We present the experimental data obtained. The experiments were run on a machine running Ubuntu 15.04 with qemu-.

We transferred 2 GB of data from a QEMU-KVM guest to the host using our framework with varying block sizes. The results are shown in Fig. 3a. As the bock size increases for the same total amount of data transferred the locking overhead goes down, leading to quicker transfers as can be seen from the plots.

We then wanted to compare transfer speeds using our framework, sockets and using an mmapped file on disk (an SSD in this case) for transfers. The

```
while(i < to_transfer) {
  // Check shared variable
  shm_acquire_cv(cvid, uid);
  if (*((bool*)offset)) {
    // Data available, read here
    memcpy(data, (void*)((bool*)offset + 1),
      size - sizeof(bool));
    ++i;
    *((bool*)offset) = false;

    // Notify the writer
    shm_notify_cv(cvid, uid);
  } else {
    // Wait for the writer
    shm_wait_cv(cvid, uid);
  }
  shm_release_cv(cvid, uid);
}
```

Figure 2: A Bare Bones Reader Loop

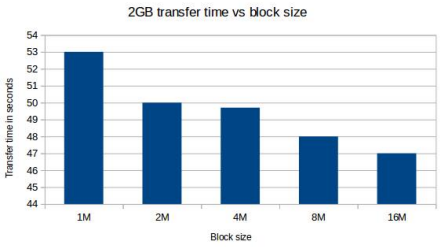
results are plotted in Fig. 3b. As can be seen, on the host, socket transfers are quicker than our framework and mmapping a shared file. This is possibly due to highly optimized code using shared memory for the loopback interface. There is an observed asymmetry between the host and guest in that the times are significantly different when the direction of data transfer using our framework changes. Data transfer from the guest to host is much faster as can be seen. Also, host to guest socket transfer are much slower than shared memory transfers demonstrating the utility of the framework.

Data transfer mainly involves locking times and shared memory access times. We measured both of these separately.

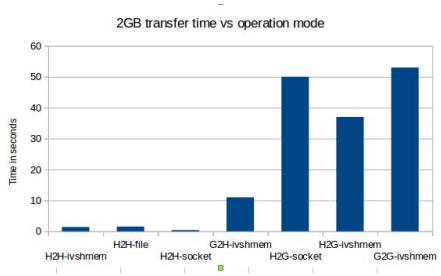
Fig, 4ashows the locking times when 20000 lock acquires and releases are performed. These are the same number of locking operations needed to transfer 2GB of data when the block size is 4MB. As can be locking overheads are a significant part (almost 20%) of the transfer times.

Shared memory access times are shown in Fig, 4b. Accessing the IVSHMEM shared memory in the guest is much slower than accessing an mmapped file in the guest. We have not yet been able to determine why this might be

the case.

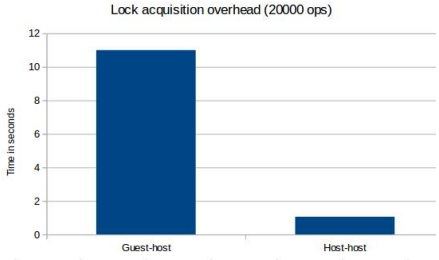


(a) Transfer time vs block size

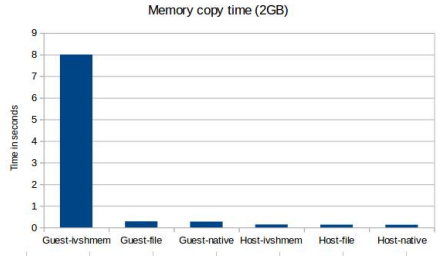


(b) Transfer time vs operation mode

Figure 3



(a) Locking overhead



(b) Shared memory access times

Figure 4

6 Future Work

We recognize the following directions for further work on the current framework.

1. **Heartbeats** Currently, if a client crashes there is a chance of memory leaks and/or dangling locks. Implementing heartbeats to check the liveness of hosts would help prevent this.

2. **Guest Memory Access** An investigation into the lower than expected transfer speeds (due to low memory access speeds) would be helpful.

3. **Optimized locking** The plot in Fig. 4a show how locking is a significant overhead when transferring data between the guest and the host. It would be great to explore ways to make this faster and reduce the transfer times significantly.

7 Conclusions

The current framework thus demonstrates the advantages of shared memory communication over socket communication, when large amounts of data is to be transferred between guests or between a guest and the host. It is hoped that with further work, the framework can be improved to achieve higher data transfer speeds.