Virtual Machine Live Migration

107021129 黃明瀧

Cloud Computing Homework 1

A. VM Setup Description

The experiments throughout this document were all done in a custom environment on my local machine. There are two "hosts"[1], analogous to host 1 and host 2 mentioned in the tutorial slides, which are themselves containers running under the LXC container runtime. The QEMU guest virtual machine to be live-migrated runs within these containers. To make things simple (and lightweight), both hosts and the guest are running the Alpine Linux distribution. The following steps were taken to setup the environment and the VM.

1. Launch containers.

```
lxc launch image:alpine:3.15 host1
lxc launch image:alpine:3.15 host2
```

2. Add a shared mount (where the VM disk image is to be put) and passthrough the KVM device to both hosts.

```
lxc storage volume create default nfs
lxc config device add host1 nfs disk \
    pool=default source=nfs path=/mnt/nfs
lxc config device add host2 nfs disk \
    pool=default source=nfs path=/mnt/nfs
lxc config device add host1 kvm unix-char path=/dev/kvm
lxc config device add host2 kvm unix-char path=/dev/kvm
```

3. Configure bridge and TAP interfaces in both hosts. To do so, edit `/etc/network/interfaces` like this.

```
auto tap0 inet manual
        pre-up tunctl -t tap0

auto br0
iface br0 inet dhcp
        bridge-ports eth0 tap0
        bridge-stp 0

hostname $(hostname)
```

   Run `rc-service networking restart` to reload.

[1] All uses of the word "host"in this document, unless otherwise specified, are referring to *host 1* and *host 2* running as containers on the actual host.

4. Launch QEMU to perform VM installation. Here the `-curses` option is used to show the TTY in a curses-based interface inside the terminal. This avoids the headaches of doing VNC and X11 forwarding[2].

   ```
   qemu-system-x86_64 -cpu host -enable-kvm -m 1G -smp 1 \
       -drive if=virtio,format=raw,file=/mnt/nfs/alpine.img \
       -boot d -cdrom alpine-virt-3.15.4-x86_64.iso \
       -curses -nographic \
       -netdev tap,id=tap0,ifname=tap0,script=no,downscript=no \
       -device virtio-net-pci,netdev=tap0
   ```

   Installation is straightforward:

   1. Login on TTY as `root`.
   2. Run `setup-alpine`.
   3. Accept the default settings all the way down. Set root password.
   4. Select `vda` as the target with the `sys` disk mode.
   5. Reboot.

   Additionally, change `/etc/ssh/sshd_config` to allow root login.

5. The VM is now in good shape and ready to perform live migration experiments.

B. CPU Performance with and without KVM Enabled

As background information, the data is obtained on a laptop running NixOS with a R7-4750U processor. Sysbench are ran for 10 seconds (with the default options) on all configurations.

- On guest **with** KVM enabled: 18678 events
- On guest **without** KVM enabled: 3621 events
- On host: 18651 events

The guest VM runs over 5 times slower with KVM disabled. This is caused by the fact that without KVM, the QEMU process has no access to the virtulization features provided by the kernel module, and thus having to rely on emulation by software. In contrast, the guest with KVM enabled performs almost identical to the host machine.

C. Network Performance with and without Virtio

The `iperf` server instance is hosted on the actual host (and not on the LXC containers).

[2] My machine is on Wayland, which complicates X11 forwarding even more.

The result with virtio:

```
[  5] 0.00-10.00 sec 19.0 GBytes 16.3 Gbits/sec 0 sender
[  5] 0.00-10.00 sec 19.0 GBytes 16.3 Gbits/sec   receiver
```

The result without virtio:

```
[  5] 0.00-10.01 sec 970 MBytes 813 Mbits/sec 0 sender
[  5] 0.00-10.01 sec 970 MBytes 813 Mbits/sec   receiver
```

From the figures, we can see that the network throughput with virtio is over 20 times faster than the other setup.

QEMU defaults to emulating the `e1000` network device for the guest, which uses the *full virtualization* technique. In contrast, the Virtio device uses the *para-virtualization* technique, which requires the guest to be aware that it's a VM and to use that virtio drivers to talk to the host. The para-virtualized drivers are designed to reduce the number of switches between the VM and the VMM, thus achieving much better performance.

Out of curiosity, I also tested the network performance with the `-R` (reverse) option, where the server sends and the client receives.

The reversed result with virtio:

```
[  5] 0.00-10.00 sec 21.1 GBytes 18.2 Gbits/sec 0 sender
[  5] 0.00-10.00 sec 21.1 GBytes 18.2 Gbits/sec   receiver
```

The reversed result without virtio:

```
[  5] 0.00-10.00 sec 2.95 GBytes 2.53 Gbits/sec 0 sender
[  5] 0.00-10.00 sec 2.94 GBytes 2.53 Gbits/sec   receiver
```

The interesting part is that the reversed result of e1000 significantly higher than the normal one. As of the time of writing, I still have not found any satisfying explanation of this phenomenon.

D. `iperf` and `sysbench` Measurements During Migration

`sysbench` Part

The experiment is done by running the following POSIX shell script in the guest during migration.

```
while true; do
    TS=$(date +%H:%M:%S)
    PERF=$(sysbench cpu run --time=1 | awk '/events \(avg\/stddev\)/ {
        printf $3 }');
    echo $TS $PERF
done
```

```
22:55:05 1848.0000/0.00
22:55:06 1839.0000/0.00
22:55:07 1852.0000/0.00
22:55:08 1847.0000/0.00
22:55:09 1840.0000/0.00
22:55:10 1844.0000/0.00
22:55:11 1852.0000/0.00
22:55:12 1848.0000/0.00
22:55:13 1854.0000/0.00
22:55:14 1852.0000/0.00
22:55:15 1858.0000/0.00
# migration starts
22:55:16 1801.0000/0.00
22:55:17 1858.0000/0.00
22:55:18 1859.0000/0.00
22:55:19 1856.0000/0.00
22:55:20 1849.0000/0.00
22:55:21 1845.0000/0.00
22:55:22 1840.0000/0.00
22:55:23 1830.0000/0.00
22:55:24 1839.0000/0.00
22:55:25 1842.0000/0.00
```