Manual

v 1.0

NETWORK EMULATION USING
ADAPTIVE TIME DILATION

MAY 2014

Contents

1 Overview **3**
- 1.1 Project Description . . . 3
- 1.2 Adaptive Time Dilation . . . 3
- 1.3 System Architecture . . . 4
- 1.4 Virtual Time . . . 4

2 Install and Run **7**
- 2.1 Environments . . . 7
- 2.2 Prerequisites . . . 7
- 2.3 Getting the code . . . 8
- 2.4 Running virtual hosts . . . 8

Chapter 1

Overview

1.1 Project Description

Building a testbed for large-scale heterogeneous systems can be costly and inefficient. Emulation is often used to evaluate the performance of a system in a controlled environment. Time dilation allows virtual machines (VMs) to emulate higher performance than that of their physical machine. We present an approach using adaptive time dilation to emulate large-scale distributed systems composed of heterogeneous machines and OSs.

Our proposed emulation system can be used to test systems comprising heterogeneous OSs. To emulate heterogeneous OSs (e.g. Linux, FreeBSD, Windows, Cisco IOS, Junos, etc), we use QEMU-KVM that supports full virtualization.

1.2 Adaptive Time Dilation

Time dilation is a technique to slow the passage of virtual time (from the perspective of a virtual host) by a specified factor, which is referred to as time dilation factor (TDF). With time dilation, physical resources appear TDF times faster (i.e., higher performance is emulated). One second of virtual time passes for every TDF seconds of real time in virtual hosts (VHs). Therefore, time dilation enables empirical evaluation at CPU speeds that are not currently available from production hardware and larger system emulation on fewer physical hosts (PHs).

A fixed TDF can result in system underutilization. Adapting time dilation to system loads allows the system to effectively utilize its resources.

1.3 System Architecture

In order to allow VMs to run unmodified OSs, the control of time dilation is performed at the level of the QEMU-KVM hypervisor. Synchronization agents exchange TDF control messages to synchronize VHs distributed over different PHs.

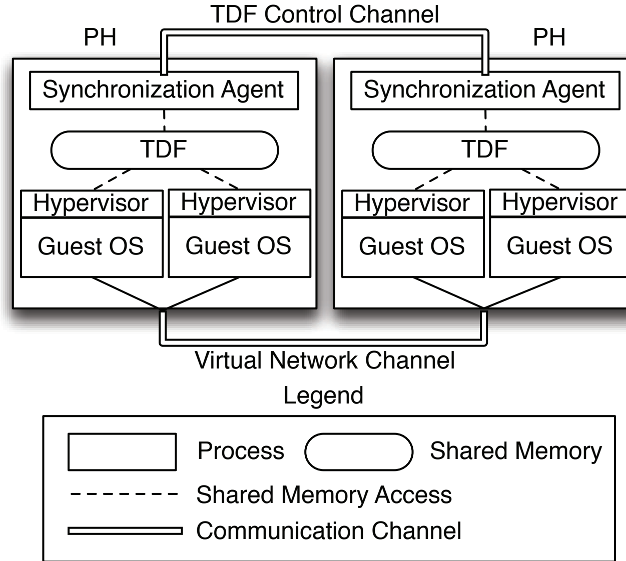

Figure 1.1: System architecture

1.4 Virtual Time

The ratio between the virtual time passage rate and real time is $\frac{1}{TDF}$. Hence, given the real time t^R, virtual time t^V can be obtained by:

$$t^V = t^V_{start}(n) + \frac{t^R - t^R_{start}(n)}{TDF(n)}, \tag{1.1}$$

where $t^V_{start}(n)$ is the value of starting point of nth TDF change (epoch) in virtual time and $t^R_{start}(n)$ is the value of starting point of nth TDF change in real time.

When our system is ready to update the TDF, it freezes the clock, changes TDF, and then restarts the clock. In our system implementation, `disable_clock()` is invoked, the variable TDF is changed, and then `enable_clock()` is invoked, as seen in Listing 1.1.

If the clock is enabled, t^V can be obtained by calling `get_virtualtime()`, where t^V_{start} is `timers_state->clock_restart + timers_state->clock_offset`, t^R is `realtime` obtained by calling `get_realtime()`, t^V_{start} is `timers_state->clock_restart`, and $TDF(n)$ is `*TDF`, as shown in Listing 1.2. In `get_realtime()`, `clock_gettime()` is a POSIX system call to retrieve the PH's real time.

If the clock is disabled, virtual time t^V is obtained by taking `timers_state->clock_offset`. The reason is that when `disable_clock()` is invoked, virtual time at that time is stored in `timers_state->clock_offset`, as seen in Listing 1.1.

```
/* Timer's state structure */
typedef struct TimersState {
    int64_t clock_offset;
    int64_t clock_restart;
    int32_t clock_enabled;
} TimersState;

/* A memory space for timers_state is created
 in shared memory. */
TimersState *timers_state;

/* Restart virtual time */
void enable_clock(void) {
    if (!timers_state->clock_enabled) {
    int64_t realtime = get_realtime();
        timers_state->clock_offset -= realtime;
        timers_state->clock_restart = realtime;
        timers_state->clock_enabled = 1;
    }
}

/* Stop virtual time */
void disable_clock(void) {
    if (timers_state->clock_enabled) {
        timers_state->clock_offset = get_virtualtime();
        timers_state->clock_enabled = 0;
    }
}
```

Listing 1.1: Enabling/disabling clocks

```
/*Memory space for TDF is created in shared memory. */
long double *TDF;

/* Get virtual time */
int64_t get_virtualtime(void) {
    int64_t virtualtime;

    for(;;){
        if (!timers_state->clock_enabled) {
            virtualtime = timers_state->clock_offset;
            if(!timers_state->clock_enabled)
                break;
        }

        if(timers_state->clock_enabled){
            int64_t realtime = get_realtime();
            virtualtime = timers_state->clock_restart
              + timers_state->clock_offset
          + (realtime-timers_state->clock_restart)/*TDF;

            if(timers_state->clock_enabled)
                break;
        }
    }

    return virtualtime;
}

/* Get real time */
int64_t get_realtime(void) {
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    return ts.tv_sec * 1000000000LL + ts.tv_nsec;
}
```

Listing 1.2: Getting virtual time

Chapter 2

Install and Run

In our emulation system, the main components are the synchronization agent and the modified hypervisor. The synchronization agents exchange TDF messages between physical hosts to synchronize virtual time for all virtual hosts. The QEMU-KVM hypervisor is modified such that virtual time at the layer of the hypervisor is generated based on the TDF, as described in Section 1.4.

2.1 Environments

Our emulation system has been tested using the following hardware and operating systems (OSs).

- Hardware for PHs: Dell PowerEdge R210
- OS for PHs: Ubuntu Linux 10.04
- OS for VHs: Ubuntu Linux 10.04, FeeBSD 9.0, Windows XP SP3, Junos 12.1

You can use the other hardware and OSs with your customization.

2.2 Prerequisites

In order to run synchronization agents. the following software packages must be installed on PHs' Linux OS.

- libboost-system1.40-dev

- libboost-thread1.40-dev
- liblog4cxx10-dev
- libgtop2-dev
- libxen3-dev → This package is needed for qemu-kvm.
- uml-utilities → This package is needed for tunctl.

These packages are required to run our synchronization agent, `virtualagentd`.

2.3 Getting the code

You can download the latest release of the synchronization agent and the modified hypervisor from our homepage where you found this manual.

1. Synchronization agent:
   - Source file: `tdfagent.1.2.tar.gz`
   - Execution file in the source file: `virtualagentd`
2. Modified hypervisor:
   - Source file: `qemu-kvm-0.13.0-atd.tar.gz`.
   - Execution file in the source file: `qemu-system-x86_64`

2.4 Running virtual hosts

In order to create virtual hosts, follow the steps below.

1. Unzip the downloaded code in your home directory.
2. Run the synchronization agent `virtualagentd` as a root in a physical host. The `virtualagentd` will run as a daemon process in the physical host. Note that each physical host run a synchronization agent.
3. Now you can create a VH using our modified hypervisor `qemu13-sync`, as shown in Listing 2.1.

```
# qemu-kvm-0.13.0/x86_64-softmmu/qemu-system-x86_64
-enable-kvm
-no-kvm-pit
-m 1024
-hda /root/images/vl0001.vdi
-net nic,vlan=0,macaddr=02:00:02:00:01:00
-net user,vlan=0,hostfwd=tcp::11001-:22,
hostfwd=tcp::11101-:13208
-net nic,vlan=1,macaddr=02:00:02:00:01:01
-net tap,vlan=1,ifname=vl0001-01,
script=/root/netemu/bin/ifup,
downscript=/root/netemu/bin/ifdown
-net tap,vlan=1,ifname=vl0001-01m,
script=/root/netemu/bin/ifup,
downscript=/root/netemu/bin/ifdown
-vnc 0.0.0.0:01
&
```

Listing 2.1: A sample command for creating a VH