InfraSIM Documentation

Release 2.0

EMC

December 07, 2016

Contents

InfraSIM allows you to deploy virtualized infrastructures consisting of simulated servers, storage devices, switches and smart PDUs(Power Distribute Units). You can use it to create development environments that simulate the exact physical environments where your product will eventually be deployed.

The project is a collection of libraries and applications housed at https://github.com/InfraSIM/ and available under the Apache 2.0 license (or compatible sublicenses for library dependencies). The code for InfraSIM is a combination of python, shell and C, etc.

CHAPTER 1

InfraSIM™ Overview and quick start video

VIDEO: InfraSIM On YouTube

1.1 Contents

1.1.1 Overview

InfraSIM provides the technology to simulate the interface and behavior of hardware devices including compute, storage, networking, and smart PDU(Power Distribute Units). It leverages the technology of virtualization which enables to simulate a big amount of hardware devices with limited physical resources. And these simulated hardware devices can be configured to construct an @scale infrastructure.

Data Center element simulating

At single node level, InfraSIM provides:

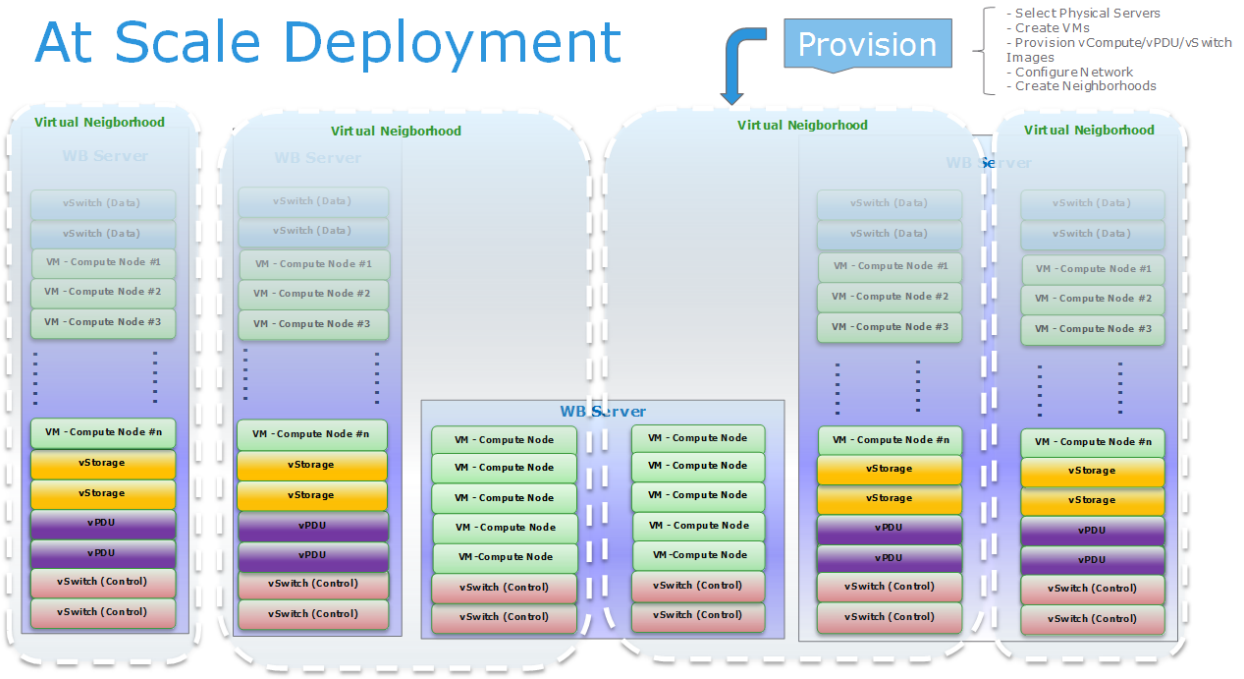
- **Precisely-simulating of bare-metal hardware node:** Server, PDU hardware configuration, manufacture information, vendor-specific interfaces and functionalities, etc
- **Mechanism for customizing sub-component of node.** i.e. Configuration and properties of Drive, NIC and processors; enclosure management subsystem.
- **Configuring and manipulating platform firmware - BIOS, POST and BMC - behavior**
- **Easy way to simulating hardware failure**

Virtual infrastructure powered by InfraSIM

To simulate scale out infrastructure, InfraSIM allows setting up, configuring one heterogeneous hardware infrastructure, with blow advantages, which provides a total solution for CI application development and test.

- Network topology simulation
- Automatic deployment on demand
- Optimized footprint - Large scale deployment on top of limited number of servers

The diagram below illustrates the development concept of the @Scale deployment.



1.1.2 Why InfraSIM?

InfraSIM provides effective, economic way to simulate a bare-metal infrastructure on which engineering team can leverage to achieve purpose of:

- Cost saving by simulating a scaled infrastructure with limited hardware materials
- Less dependency on hardware material which is in short of
- Increase automation level and eventually increase development and testing efficiency
- Increase test coverage by leveraging InfraSIM error injection functionality

There're many existing virtualization technologies like VMWare product, KVM, XEN, etc. They are aiming at provisioning generic virtual machines which contains computing power and storage capacities and networking functionalities. However, they're not sufficient in many engineering areas because of several missing pieces like:

- **Vendor personality:** e.g. vendor identification information, SKU information, MFG information. No way to tell it is Cisco switch or brocade one; or tell it is dell server or IBM server;
- **Vendor-specific functionality:** e.g. Dell Remote Access Controller; Cisco UCS appliance – provided central interface for management
- **Rack/Chassis/PDU/PS/Cooling:** fundamental building blocks of hardware. Particular software need to be aware of these info and would need to do some analytic and decision-making by checking details and running status of these components.
- **Platform FW behavior:** VM provided only limited number of adjustable parameters; Limited emulation to FW behaviors.

Virtual machines directly spawn by virtualization technologies are designed in way to be working forever, which might not be totally expected in some situations. There definitely are desires to simulate hardware failures to test

robustness/error recovery scheme of software.

So we can conclude easily that current popular virtualization technologies are not designed to precisely simulate hardware and consequently it can't be directly adopted in dev and validation activities, for software for purpose of infrastructure and hardware management and orchestration, which really have dependencies on detailed hardware properties.

While InfraSIM is really designed to precisely simulate hardware and bare-metal infrastructure in order to maximize the productivity and flexibility of you and your team.

InfraSIM Use Cases

Currently, InfraSIM has successfully proved that it is capable of not only saving lots of cost of purchasing hardware material for setting up a pure bare-metal environment, but also providing many flexibilities in software developing and testing areas. Here're 2 cases where InfraSIM is leveraged for software application - RackHD and VMWare software - development and testing.

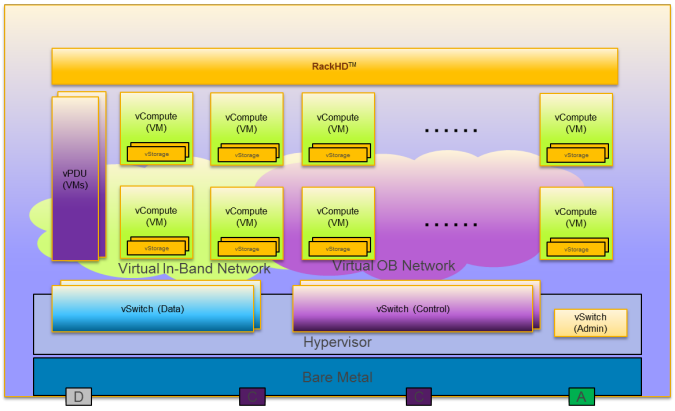
InfraSIM as test infrastructure of RackHD™

Notes: RackHD™ is an open source project that provides hardware orchestration and management through RESTful APIs. For more information about RackHD, go to http://rackhd.readthedocs.io.

1. **At scale test**

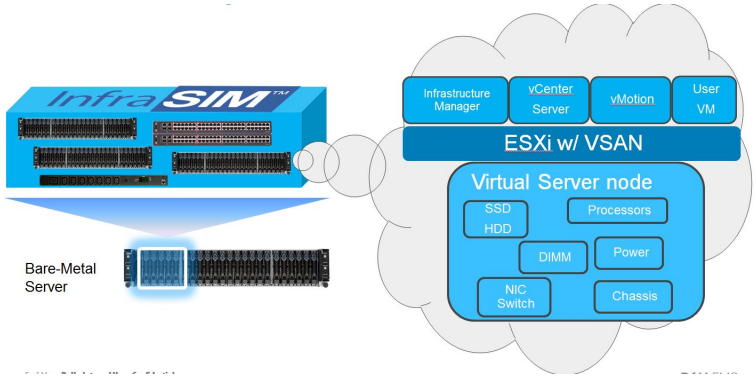
   We can validate RackHD functionalities by having it manage and orchestrate a virtual infrastructure with adjustabl

   - Big number of nodes
   - Diversity of node type - different type, model, vendors, etc
   - Increased complexity of network topology



2. **Telemetry data testing** InfraSIM allows generating and modifying server sensor readings that we can better test feature of telemetry data of RackHD.
3. **Node provision** InfraSIM allows customizing node device tree and manipulating FE behavior, we can better test node provision feature, for example, bootstrapping servers and deploying operating systems, hypervisors and applications.
4. **Error injection** Because InfraSIM is adopting software approach to simulate hardware, both elements and entire infrastructure, it provided more feasibility and easiness to simulate hardware failures to test our software error handing logic.

Running VMWare Virtualization Software inside InfraSIM Virtual server



1.1.3 Installation

For virtual node simulating server or PDU, either bare-metal machine (server, laptop, desktop) or virtual machine can host one. It requires configuring network (either corresponding physical one or virtual one, even mixing physical and virtual network together for hybrid configuration) in order to compose one infrastructure containing virtual servers, PDUs and specified network topology.

Here's requirement on hardware environment and virtualization environment running InfraSIM:

Requirement

Pre-requisite

Several mandatory configuration has to be made as below which is required to accommodate InfraSIM virtualization-nesting design. How to install VMWare ESXi describes a example of how to achieve them when installing and configuring VMWare ESXi.

1. Virtual InfraSIM servers runs in the best performance if hardware-assisting technology has been enabled on underlying physical machines. These technology includes VT-d feature and AMD-V for processors from Intel and AMD.

   Note: **Physical machine** - enable VT-d in BIOS

2. When virtual server is running inside VM, it also requires underlying hypervisor passing down the hardware-virtualization-assisting to virtual machine it spawn.

   Note: **VMWare ESXi hypervisor** - Set **"vhv.enable = "TRUE"**

   Caution: **InfraSIM running on VirtualBox** will have performance penalty when running specific work load (deploying operating system, running compute-intensive application inside virtual server). This is because VirtualBox doesn't support simulating a platform which is capable of supporting hardware-virtualization-assisting feature.

3. Ensure Promiscuous Mode of virtual switch, virtual network controller has been enabled for underlying hypervisors hosting virtual machines running InfraSIM inside. Here's example on how to achieve it on top VMWare ESXi

Note: **Promiscuous Mode** - How to install VMWare ESXi

Resource Requirement

- 1 physical CPU or 1 virtual CPU
- 4GB memory
- 16GB disk space
- 1 virtual or physical NIC

Software environment

Ubuntu Linux 64-bit - 16.04 is recommended

Virtual Server

1. Ensure sources.list integrity then install dependency:

```
sudo apt-get update
sudo apt-get install python-pip libpython-dev libssl-dev
```

2. Upgrade pip and install setuptools:

```
sudo pip install --upgrade pip
sudo pip install setuptools
```

3. Select either one of below ways to install infrasim:
   - install infrasim from source code:

```
git clone https://github.com/InfraSIM/infrasim-compute.git
cd infrasim-compute
sudo pip install -r requirements.txt

sudo python setup.py install
```

   - install infrasim from python library:

```
sudo pip install infrasim-compute
```

1.1.4 Getting started

This chapter describes how to access virtual server, virtual PDU and virtual infrastructure provided by InfraSIM.

Quick start of infrasim-compute application

Command interfaces

1. Initialization (you need do it once)

```
sudo infrasim-init
```

2. Start Infrasim Service:

```
sudo infrasim-main start
```

Verify your service by *VNC and IPMI*

3. Status and version number check:

```
sudo infrasim-main status
sudo infrasim-main version
```

4. Stop Infrasim Service:

```
sudo infrasim-main stop
```

5. Start IPMI Console:

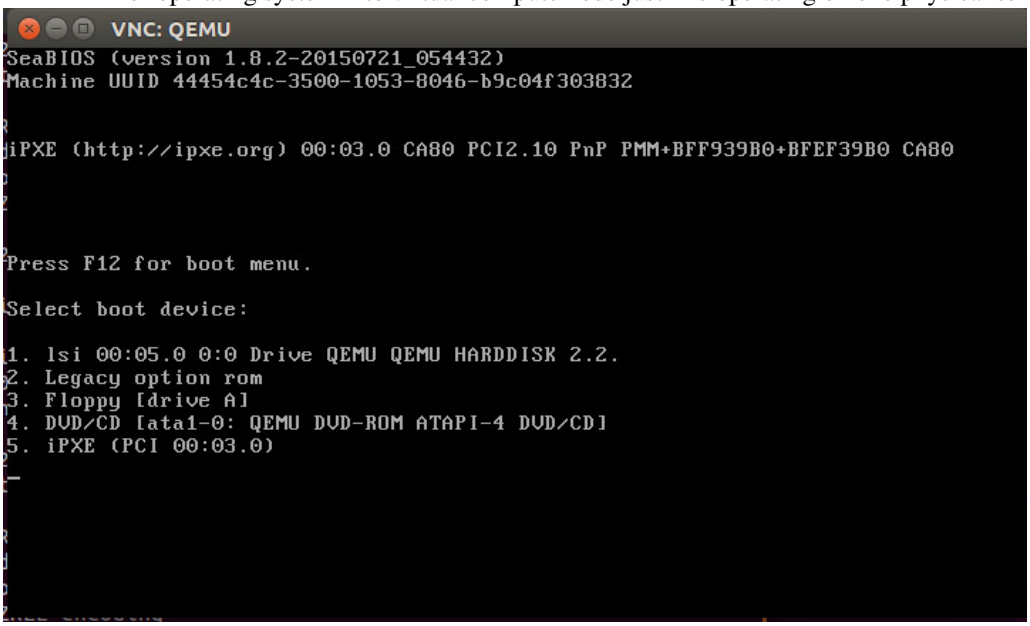
```
sudo ipmi-console start
```

6. Stop IPMI Console:

```
sudo ipmi-console stop
```

Interface to access virtual server

1. **Server graphic UI** VNC service is available through port **5901**. You can see the virtual monitor is already running and listing boot devices of virtual node. Through this booting devices, you can deploy hypervisor or operating system into virtual compute node just like operating on one physical server



2. Virtual BMC

- Install ipmitool on host machine.:

```
sudo apt-get install ipmitool
```

IPMI over LAN:

```
ipmitool -I lanplus -U admin -P admin -H <IP address> sdr list
```

Note: <IP address> is address of NIC assigned to BMC access in YAML configuration file

IPMI over internal path (vKCS) which requires OS and ipmitool application deployed inside virtual server:

```
ipmitool sdr list
```

You can get the command result like the following

```
        Pwr Unit Status  | Not Readable      | ns
        IPMI Watchdog    | Not Readable      | ns
        FP NMI Diag Int  | Not Readable      | ns
        SMI TimeOut      | Not Readable      | ns
        System Event Log | Not Readable      | ns
        System Event     | Not Readable      | ns
...
```

1. Serial over LAN

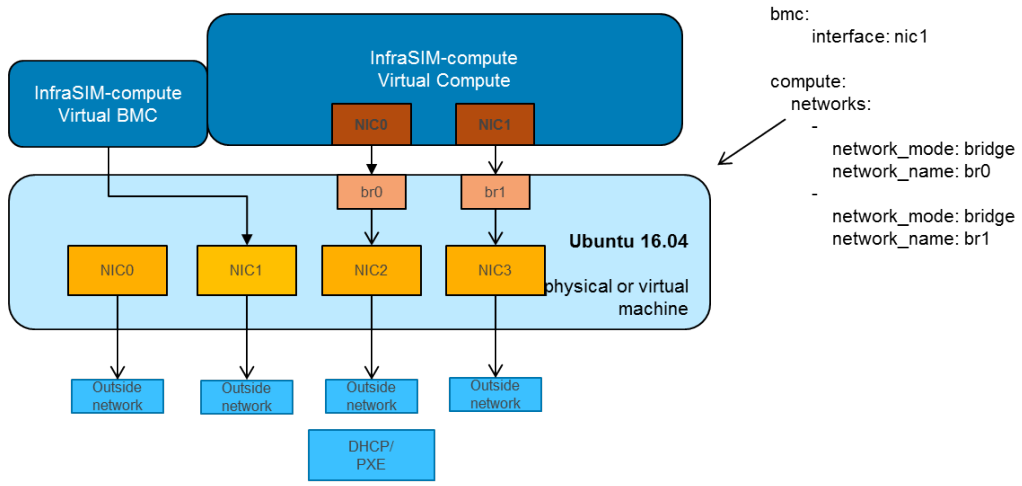
   It requires activate SoL through IPMI command and console running IPMI console will becomes serial console of virtual server. After InfraSIM services started, this command is to activate SoL:

```
sudo ipmitool -I lanplus -U admin -P admin -H localhost sol activate
[SOL Session operational.  Use ~? for help]
```

Setup an InfraSIM Virtual Server on ESXi

To setup an InfraSIM Server on ESXi, you should have an OVA with necessary environment prepared. You can consult the InfraSIM team to get the image or build one with the packer build image. Below are the steps to deploy and run InfraSIM on ESXi:

1. Get ESXi environment prepared by following instruction
2. Spin up a virtual machine by choosing "Deploy OVF Template". Specify the URL of the OVA image.
3. Map the networks used in the OVA. The networking configured inside OVA is multi-bridge mode:



4. Modify YAML configuration file as you need. The default configuration for OVA is infrasim.yml. The path is:

```
/usr/local/infrasim/etc/infrasim.yml
```

5. Kick off all InfraSIM services.
6. Done, enjoy this virtual server!

Note: No need to run **infrasim-init** because it's already done during image build.

Configuration for OVA can be refered on Packer OVA Configuration. Below are the major parameters:

```
Disk Size: 40G
Memory: 8G
Number of CPUs: 2
Number of NICs: 4
Type of NICs: VMXNET 3
NIC0:
    Name: ens160
    networkName: ADMIN
NIC1:
    Name: ens192
    networkName: BMC
NIC2:
    Name: ens224
    networkName: CONTROL
    Promiscuous Mode: on
NIC3:
    Name: ens256
    networkName: DATA
    Promiscuous Mode: on
```

Setup an InfraSIM Virtual Server in VirtualBox

Virtualbox is available on multiple platforms. To get an InfraSIM BOX image, refer to packer build image

1. Install virtualbox on the host.
2. Create a directory for the VM and move the BOX image along with Vagrantfile under the directory.
3. CD to the directory and run commands:

```
    vagrant box add --name infrasim-compute <YOUR_BOX_IMAGE>
    vagrant up
    vagrant ssh
```

4. Modify YML configuration if you need.
5. Start InfraSIM services. No **"infrasim-init"** needed.

BOX configuration can be refered on Packer BOX Configuration and Vagrantfile. The major parameters are:

```
Disk Size: 40G
Memory: 5G
Number of CPUs: 2
Number of NICs: 4
NIC0:
    Name: enp0s3
    Network Adapter: NAT
NIC1:
    Name: enp0s8
    Network Adapter: Internal Network
NIC2:
    Name: enp0s9
    Network Adapter: Internal Network
    Promiscuous Mode: on
NIC3:
    Name: enp0s10
    Network Adapter: Bridged Adapter
    Promiscuous Mode: on
```

1.1.5 Configuration

Virtual Server Configuration file

There's one central virtual server configuration file which is **/etc/infrasim/infrasim.yml** (source code). All adjustable parameters are defined in this file. This is the only file to modify if you want to customize or make adjustment on the virtual server node. While not all supported options are explicitly listed in this file for purpose of simplicity. However there's one example configuration file - **/etc/infrasim.full.yml.example** (source code) - listed all supported parameters and definitions. By referring content in example file, you can modify real file infrasim.yml and then restart infrasim-main service and then new properties will take effect.

Here's full list of the example configuration file; every single key-value pair is supported to be add/modify in your real-in-use infrasim.yml:

```
# This example virtual server configuration file intends to throughout
# list parameters and properties that infrasim-compute virtual server
# supports to adjust. In most cases it is fine to use default value
# for particuar configuration by skipping putting it into infrasim.yml
# configuration file. For anything item you're interested, it is recommended
# to look up infomation here first. For example, if you'd like to customize
# properties of your drive - either serial number or vender - in below there're
# corresponding item to show how to achieve that.

# Unique identifier
name: node-0

# Node type is mandatory
# Node type of your infrasim compute, this will determine the
```

```
# bmc emulation data and bios binary to use.
# Supported compute node names:
#  quanta_d51
#  quanta_t41
#  dell_c6320
#  dell_r630
#  dell_r730
#  dell_r730xd
#  s2600kp - Rinjin KP
#  s2600tp - Rinjin TP
#  s2600wtt - Node of Hydra, Python
type: quanta_d51

compute:
    # n - Network (PXE); c - CD-ROM;
    # d - Drive (bootindex in drive sections controls order of booting HDD)
    boot_order: ncd
    kvm_enabled: true
    numa_control: true
    cpu:
        model: host
        features: +vmx
        quantities: 8
    memory:
        size: 4096
    # Currently the PCI bridge is only designed for megasas storage controller
    # When you create multiple megasas controller, the controllers will be assigned
    # a different pci bus number
    pci_bridge_topology:
        -
            device: i82801b11-bridge
            addr: 0x1e.0x0
            multifunction: on
                -
                    device: pci-bridge
                    chassis_nr: 0x1
                    msi: false
                    addr: 0x1
    storage_backend:
        -
            controller:
                type: ahci
                max_drive_per_controller: 6
                drives:
                -
                    model: SATADOM
                    serial: HUSMM142
                    bootindex: 1
                    # To boot esxi, please set ignore_msrs to Y
                    # sudo -i
                    # echo 1 > /sys/module/kvm/parameters/ignore_msrs
                    # cat /sys/module/kvm/parameters/ignore_msrs
                    file: chassis/node1/esxi6u2-1.qcow2
                -
                    vendor: Hitachi
                    model: HUSMM0SSD
                    serial: 0SV3XMUA
                    # To set rotation to 1 (SSD), need some customization
```

```
                    # on qemu
                    # rotation: 1
                    # Use RAM-disk to accelerate IO
                    file: /dev/ram0
                -
                    vendor: Samsung
                    model: SM162521
                    serial: S0351X2B
                    # Create your disk image first
                    # e.g. qemu-img create -f qcow2 sda.img 2G
                    file: chassis/node1/sda.img
                -
                    vendor: Samsung
                    model: SM162521
                    serial: S0351X3B
                    file: chassis/node1/sdb.img
                -
                    vendor: Samsung
                    model: SM162521
                    serial: S0451X2B
                    file: chassis/node1/sdc.img
        -
            controller:
                type: megasas-gen2
                use_jbod: true
                use_msi: true
                max_cmds: 1024
                max-sge: 128
                max_drive_per_controller: 1
                drives:
                    -
                        vendor: HITACHI
                        product: HUSMM168XXXXX
                        serial: SN0500010351XXX
                        rotation: 1
                        slot_number: 0
                        wwn: 0x50000ccaxxxxxxxx
                        file: <path/to/your disk file>

    networks:
        -
            network_mode: bridge
            # Bridge need to be prepared beforehand with brctl
            network_name: br0
            device: vmxnet3
            mac: 00:60:16:9e:a8:e9
        -
            network_mode: nat
            device: e1000
    ipmi:
        interface: bt
        chardev:
            backend: socket
            host: 127.0.0.1
            reconnect: 10
        ioport: 0xca8
        irq: 10
    smbios: chassis/node1/quanta_d51_smbios.bin
```

```
    monitor:
        mode: control
        chardev:
            backend: socket
            server: on
            wait: off
            path: <path/to/your/sock file>
    # set vnc display <X>
    vnc_display: 1
bmc:
    interface: br0
    username: admin
    password: admin
    address: <ip address>
    channel: 1
    lancontrol: <path/to/lan control script>
    chassiscontrol: <path/to/chassis control script>
    startcmd: <cmd to be excuted>
    startnow: true
    poweroff_wait: 5
    kill_wait: 5
    historyfru: 20
    config_file: <path/to/your config file>
    emu_file: chassis/node1/quanta_d51.emu
    ipmi_over_lan_port: 623

# SSH to this port to visit ipmi-console
ipmi_console_ssh: 9300

# Renamed from telnet_listen_port to ipmi_console_port, extracted from bmc
# ipmi-console talk with vBMC via this port
ipmi_console_port: 9000

# Used by ipmi_sim and qemu
bmc_connection_port: 9100

# Used by socat and qemu
serial_port: 9003
```

Up to infrasim-compute commit ef289c55

- **name**

  This attribute defines nodes name, which is a unique identifier for infrasim-compute instances on the same platform. More specifically, it is used as workspace folder name.

  NOT Mandatory

  Default: "node-0"

  Legal Value: String

- **type**

  This attribute defines supported nodes type in InfraSIM. With this attribute, infrasim-compute will set BMC emulation data for `ipmi_sim` and BIOS binary for `qemu` accordingly, you can get corresponding .emu and .bin in `/usr/local/etc/infrasim/` by default.

  Mandatory

  Legal Values:

  - "quanta_d51"
  - "quanta_t41"
  - "dell_c6320"
  - "dell_r630"
  - "dell_r730"
  - "dell_r730xd"
  - "s2600kp", for Rinjin KP
  - "s2600tp", for Rinjin TP
  - "s2600wtt", for Hydra, Python

- **compute**

  This block defines all attributes used by qemu. They will finally be translated to one or more qemu command options. The module `infrasim.model.CCompute` is handling this translation. This is much like a definition for libvert, but we may want it to be lite, and compatible with some customized qemu feature in InfraSIM.

- **compute:boot_order**

  This attribute defines boot order for `qemu`. Will be translated to `-boot {boot_order}`.

  Not Mandatory

  Default: "ncd", means in a order of pxe > cdrom > default.

  Legal Value: See `-boot` in qemu-doc.

- **compute:kvm_enabled**

  This attribute enable kvm when you announce it as True and your system supports kvm. It will be translated to `--enable-kvm`. You can check if your system supports kvm by check if `/dev/kvm` exists.

  Not Mandatory

  Default: Depends on if `/dev/kvm` exists.

  Boolean Table

kvm_enabled	/dev/kvm	–enable-kvm
true	yes	yes
true	no	no
false	yes	no
false	no	no
not define	yes	yes
not define	no	no

- **compute:numa_control**

  This attribute enable NUMA to improve InfraSIM performance by binding to certain physical cpu. If you have installed `numactl` and set this attribute to True, you will run qemu in a way like `numactl --physcpubind={cpu_list} --localalloc`.

  Not Mandatory

  Default: Disabled

- **compute:cpu**

This group of attributes set qemu cpu characteristics. The module `infrasim.model.CCPU` is handling the information.

- **compute:cpu:model**

  This attribute sets qemu cpu model.

  Not Mandatory

  Default: "host"

  Legal Values: See `-cpu model` in qemu-doc.

- **compute:cpu:features**

  This attribute adds or removes cpu flags according to your customization. It will be translated to `-cpu Haswell,+vmx` for example.

  Not Mandatory

  Default: "+vmx"

  Legal Values: See `-cpu model` in qemu-doc.

- **compute:cpu:quantities**

  This attribute sets virtual cpu numbers in all. With default socket 2, CCPU calculates core per socket. Default set to 1 thread per cores. It will be translated to `-smp {cpus},sockets={sockets},cores={cores},threads=1` for example.

  Not Mandatory

  Default: 2

  Legal Values: See `-smp` in qemu-doc.

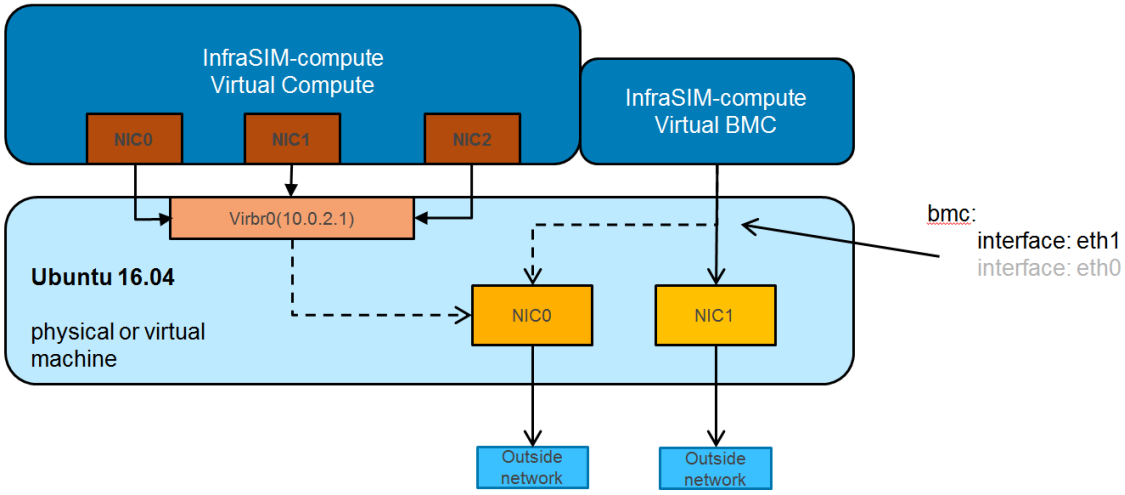
- **compute:memory**
- **compute:memory:size**
- **compute:storage_backend**
- **compute:storage_backend:-:controller**
- **compute:storage_backend:-:controller:type**
- **compute:storage_backend:-:controller:max_drive_per_controller**
- **compute:storage_backend:-:controller:use_jbod**
- **compute:storage_backend:-:controller:use_msi**
- **compute:storage_backend:-:controller:max_cmds**
- **compute:storage_backend:-:controller:max-sge**
- **compute:storage_backend:-:controller:drives**
- **compute:storage_backend:-:controller:drives:-:model**
- **compute:storage_backend:-:controller:drives:-:serial**
- **compute:storage_backend:-:controller:drives:-:bootindex**
- **compute:storage_backend:-:controller:drives:-:file**
- **compute:storage_backend:-:controller:drives:-:vendor**
- **compute:storage_backend:-:controller:drives:-:rotation**

- **compute:networks**
- **compute:networks:-:network_mode**
- **compute:networks:-:network_name**
- **compute:networks:-:device**
- **compute:networks:-:mac**
- **compute:ipmi**
- **compute:ipmi:interface**
- **compute:ipmi:chardev**
- **compute:ipmi:chardev:backend**
- **compute:ipmi:chardev:host**
- **compute:ipmi:chardev:reconnect**
- **compute:ipmi:ioport**
- **compute:ipmi:Irq**
- **compute:smbios**
- **compute:monitor**
- **compute:monitor:mode**
- **compute:monitor:chardev**
- **compute:monitor:chardev:backend**
- **compute:monitor:chardev:server**
- **compute:monitor:chardev:wait**
- **compute:monitor:chardev:path**
- **compute:vnc_display**
- **bmc**
- **bmc:interface**
- **bmc:username**
- **bmc:password**
- **bmc:address**
- **bmc:channel**
- **bmc:lancontrol**
- **bmc:chassiscontrol**
- **bmc:startcmd**
- **bmc:startnow**
- **bmc:poweroff_wait**
- **bmc:historyfru**
- **bmc:config_file**
- **bmc:emu_file**

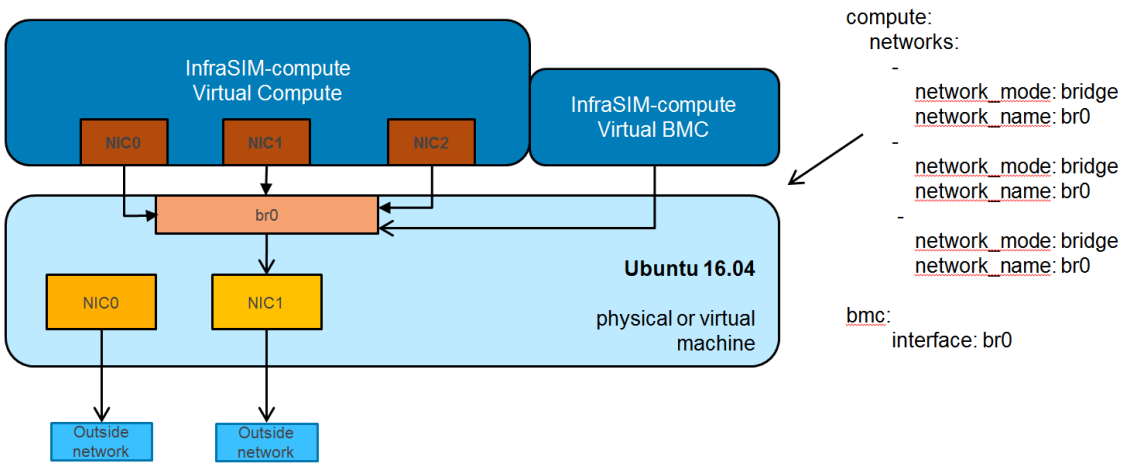
- **bmc:ipmi_over_lan_port**
- **ipmi_console_ssh**
- **ipmi_console_port**
- **bmc_connection_port**
- **serial_port**

Networking

1. **Virtual server NAT or host-only mode, this is default mode implemented in infrasim-compute**
    - vCompute is accessible ONLY inside Ubuntu host
    - Software running in vCompute can access outside network if connecting Ubuntu host NIC with virtual bridge
    - Configuration YAML file can specify which NIC IPMI over LAN traffic flows through



2. **Bridge mode - single**
    - Work as virtual switch
    - Connect BMC NIC and NICs in virtual compute together
    - Configuration YAML file controls how many NICs that virtual compute has and specify bridge they connect to

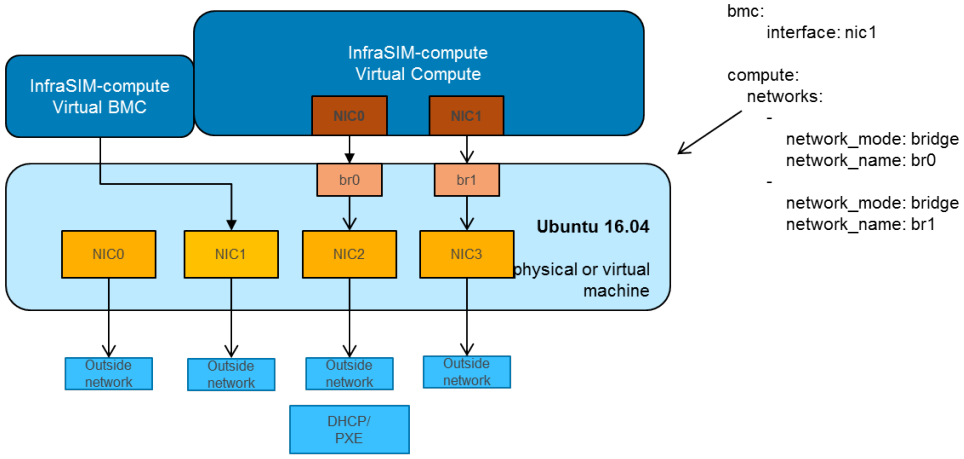


Note: It requires setting up bridge and connect to NIC of underlying host in advance.

Here's steps for this example:

```
# brctl addr br0
# brctl addif br0 eth1
# brctl setfd br0 0
# brctl sethello < bridge name > 1
# brctl stp br0 no
# ifup br0
```

3. Bridge mode - multiple



1.1.6 Software Architecture

InfraSIM Components

Below tables demonstrate the simulated hardware elements by InfraSIM.

Terminology	Description
InfraSIM	**Use the combination of hardware virtualization and emulation technol** The simulated hardware elements are called by the 'vXXX' term, with the prefix "v" for virtual.
vCompute	**Virtual Compute Node** The simulation of a physical compute node which includes the core compute subsystem and the standby BMC that control and monitor hardware resources of the compute node.
vHost	**Virtual Host (CPU subsystem)** The simulation of the core compute subsystem of a compute node. vHost is the core hardware resources of the compute node that host OS and product applications.
vBMC	**virtual BMC. It contains two concepts depending on the reference cont** 1. The simulated BMC controller of a compute node. 2. A wrapping VM image containing virtual BMC and the whole compute node implementation.
vSwitch	**Virtual Switch** The virtualized control, data, or admin switch.
vPDU	**Virtual Smart PDU** The simulation of the smart PDU.

InfraSIM uses hypervisor - either VMWare ESXi or VMWare Workstation or KVM or VirtualBox or container(docker) - to host virtual elements of infrastructure. These virtual elements are implemented inside virtual machines and consists of the following components:

- **vCompute** The virtual node is used to simulate specific server node. The virtual node component is implemented within a virtual machine running on a hypervisor.

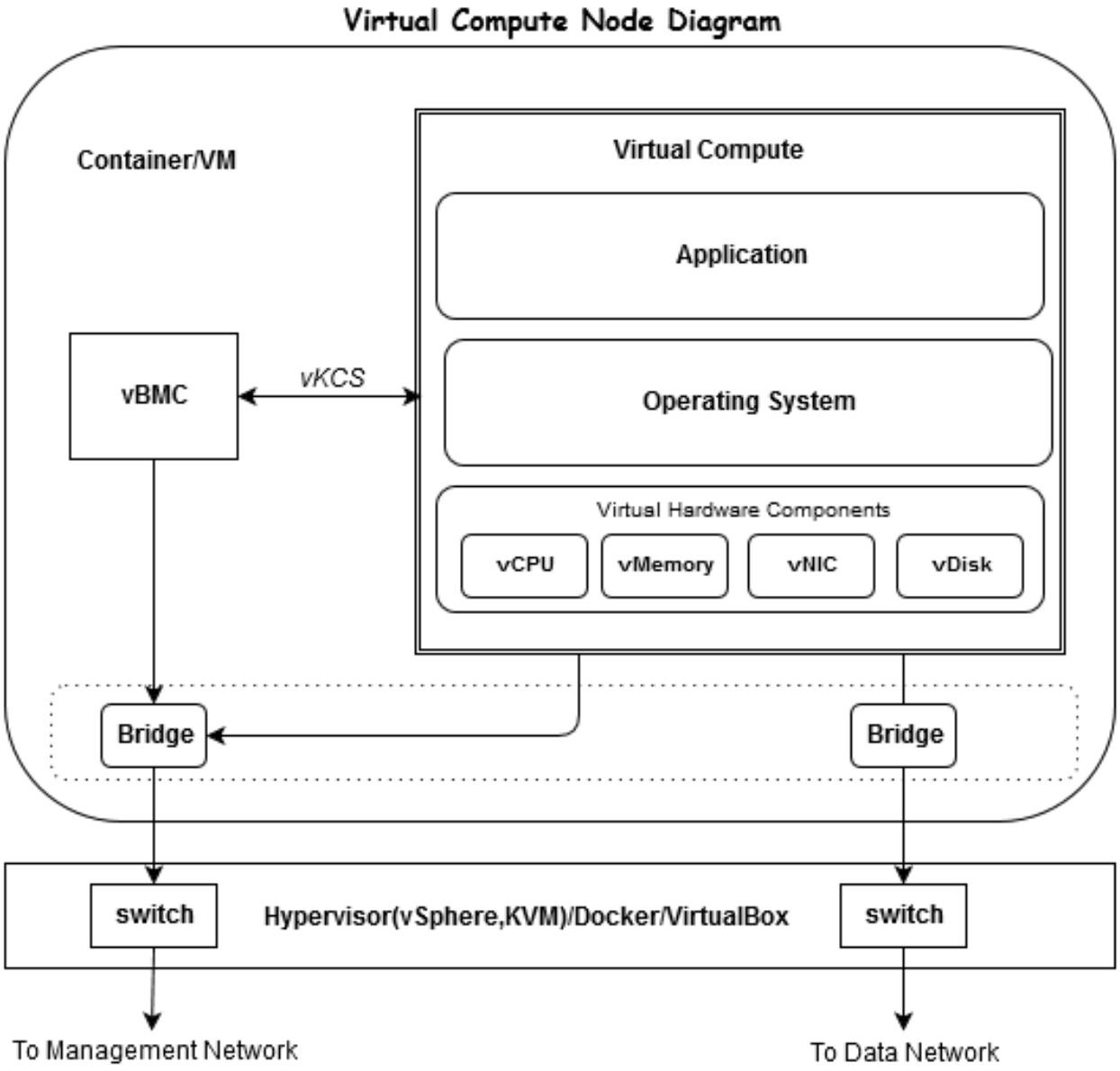
  Each virtual node implemented a virtual BMC (vBMC) inside. All BMC functionalities such as sensor data, thresholds, power controls, and boot options are simulated with this module. Both local and remote IPMI command are fully supported by using popular IPMItool.

  There's one nested QEMU VM included, which is capable of simulating CPUs, DIMMs, and other hardware devices.

- **vPDU** The vPDU is simulating intelligent PDU which is used to control AC power of other virtual nodes.
- **vSwitch** The vSwitch is used to simulate network switches, including the connections to the virtual compute nodes within the virtual infrastructure.

Virtual Node

The following diagram shows a high-level view of components in the virtual node architecture.



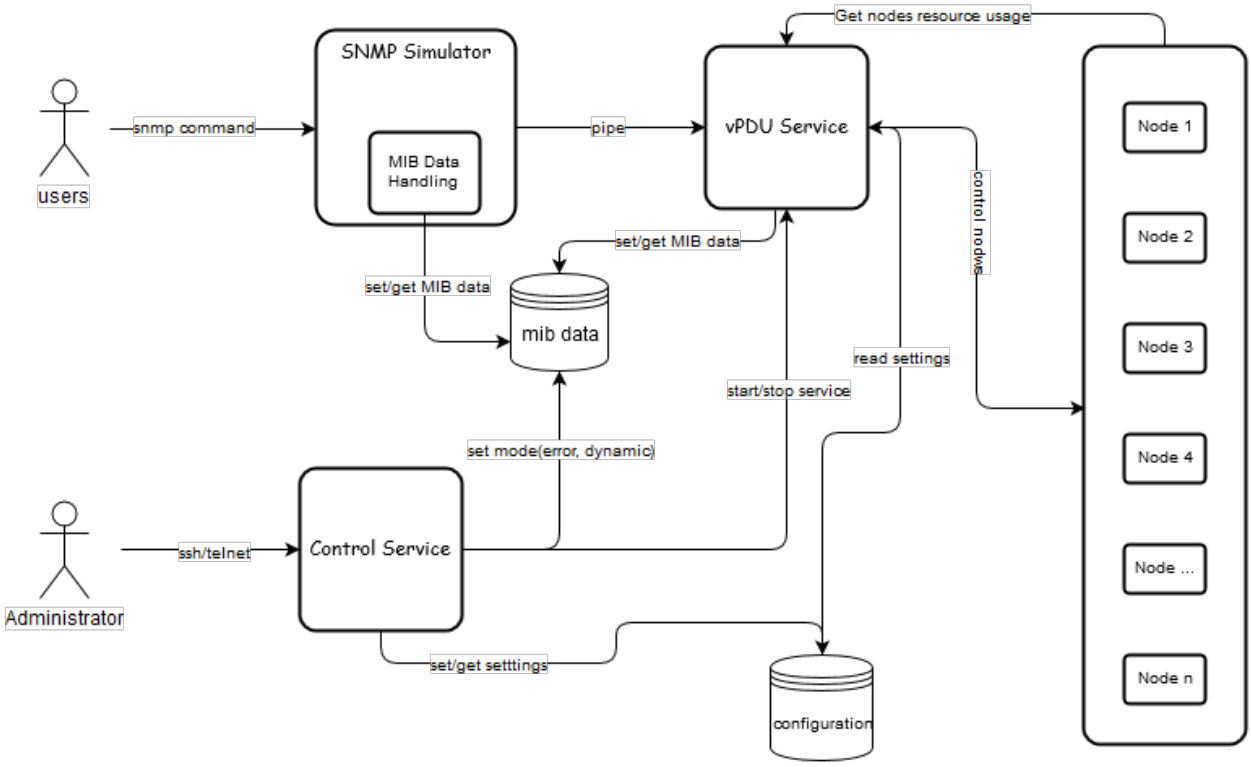
vBMC is able to handle IPMI command from either external network or local virtual compute over vKCS interface, it is bridged to an external vSwitch, to be accessible to management network.

vCompute is nested QEMU VM. There are two virtual networks attached: one is connected to the same network as vBMC which allows traffic of DHCP, TFTP, PXE, etc; the other network is used as data network specifically for user work load.

The vNode could be running on most of popular hypervisors such as VMWare ESXi, VMWare Workstation, KVM, VirtualBox, as well as container (docker).

Virtual PDU

The following diagram shows a high-level view of components in the virtual PDU architecture as well as shows how each component interacts with others.



- SNMP Simulator

  The snmp simulator is to simulate SNMP protocol, to respond snmp requests from external sources. It also supports parsing and responding according to definitions of a vendor-specific MIB data file, if you want to get more details of this simulator, please reference snmpsim

- vPDU Service

  The vPDU service will handle the messages from snmp simulator over pipe, and then call various control interface to power on, power off, reboot the virtual nodes.

- Control service

  The control service is an interface over SSH to configure vPDU such as ip address, simulated data settings, outlet settings etc.

Virtual Switch

Regarding to vSwitch solution, InfraSIM mainly leverages products from Hypervisor - for example VMWare vSwitch; or from vendor such as Cisco Nexsus 1000v, Arista vEOS.

1.1.7 Features

Bare-metal server simulation

Here's a list of physical servers that InfraSIM has simulation support:

- Dell R730XD, R630 and C6320
- Quanta T41, D51
- Intel S2600KP, S2600TP and S2600WTT

Below list all the functionalities, regarding to how InfraSIM simulates behaviors, properties of those physical server.

Virtual BMC

1. Defining channel number and support internal and remote IPMI command
    - Virtual BMC is interally accessible (similar with a virtual KCS path) through software applications - typically ipmitool - running inside virtual server
    - Virtual BMC supports IPMI over LAN
    - Supports ipmitool "-t" option and specify access channel
2. FRU, Sensors, SDR, LAN, User
    - FRU, Sensor and SDR data - simulating what corresponding physical server is presenting
    - Define Chassis/Node relation by customizing, Chassis s/n and Node slot information
3. Chassis Control
    - Power control and power status monitoring
    - Connecting to virtual host to really simulate power control behavior of a physical server
4. IPMI master read/write to simulate I2C device
    - Define and inject data for IPMI master read/write of particular I2C device
5. SEL
    - System power up event generating in SEL
    - SEL event generating on clear operation
    - SEL event generating on sensor reading beyond threshold
    - Inject SEL entry based on sensor event
    - Inject SEL entry based on OEM-defined format
6. Sensor data manipulating and injecting
    - Sensor readings dynamically change
    - Manually specify sensor (analog type) reading at run time
    - Manually specify sensor (discrete type) reading at run time
7. Supports changing boot order, activate/de-activate server Serial-Over-Lan
8. Specify NIC to transfer data for IPMI over LAN

Virtual network interface controller

1. Add, remove NIC for virtual server
2. Randomly generating MAC address for each NIC to prevent duplication
3. Supports NAT, bridge and MACVTAP modes

Virtual host

1. Support booting from PXE, ISO, HDD
2. SMBIOS data capturing and injecting
3. Define processor, memory properties

Virtual direct-attached storage

1. **Specify drive properties:**
    - SSD or spinning drive
    - Serial number
    - Physical information
    - Enable RAM disk to boost virtual disk drive performance

Intelligent power distribute unit simulation

InfraSIM has simulation for 2 types of PDU: Panduit PDU and Server Tech PDU. So far it only supports powering control virtual servers running on top of VMWare ESXi. All supported features include:

1. SNMP interface for management and control
2. Telnet/SSH Service to configure virtual PDU
3. Authentication
4. Retrieve telemetry data
5. Virtual server and outlet binding
6. Power control virtual node hosted by ESXi
7. Notification / Trap

1.1.8 User Guide

This chapter will deep to the InfraSIM usage of virtual servers, virtual PDU.

Customizing virtual Server

All supported virtual server configurations and properties of sub-component in that central configuration file. This sections describes key blocks and fileds in this YAML configuration file:

- Name of server node:

```
name: node-1
```

- Node type which specify BMC configurations and behavior (server of specific model from specific vendor) and properties defeined in SMBIOS data. Implementation behind is specifying emulation data for vitual BMC and SMBIOS to load. Then ultimate, those IPMI command and dmidecode running on virtual server will get response exactly the same as what you can get from one physical server. By default it loads emulation data of Quanta D51 type:

```
    type: quanta_d51
```

- compute - This is one big block which contains several sub-block: storage, network, ipmi
  - Storage block is also arranged in an hierarchy way by storage_backend/controller/drives; for every single drive added, InfraSIM allows defining model/serial number/vendor/media/image file:

```
        vendor: Hitachi
        model: HUSMM0SSD
        serial: 0SV3XMUA
        # To set rotation to 1 (SSD), need some customization
        # on qemu
        # rotation: 1
        # Use RAM-disk to accelerate IO
        file: /dev/ram0
```

- networks - defining network sub-system of virtual server. As below, 2 vmxnet3 type NICs are populated and connected to virtual switch br0:

```
    -
        network_mode: bridge
        network_name: br0
        device: vmxnet3
    -
        network_mode: bridge
        network_name: br0
        device: vmxnet3
```

Note: Virtual bridge need to be created manually beforehand by using brctl utility

- ipmi - support specifying NIC (from host) attached and BMC credential and emulation data file:

```
    bmc:
        interface: br0
        username: admin
        password: admin
        emu_file: chassis/node1/quanta_d51.emu
```

BMC run-time manipulating

InfraSIM implemented one IPMI console which allows manipulating BMC behavior at run time; it can be treated as backdoor of virtual BMC which is particular useful when simulating chassis abnormal conditions and failures. It includes functionalities:

- Update sensor reading with specified value, or cross-threshold value
- Generate dynamicly-changing reading for specific sensor
- Inject SEL entries for the particular sensors
- Inject SEL entries for arbitry defined format

Here's instructions on how to use InfraSIM IPMI console:

- Start ipmi console service by running command on host console:

```
    sudo ipmi-console start &
```

- Enter IPMI_SIM by below command. <vbmc_ip> is localhost if you're run command in host, otherwise it is IP address of NIC specified in configuration file for ipmi to use. Prompt means successfull connection to ipmi console:

```
ssh <vbmc_ip> -p 9300
IPMI_SIM>
```

- Enter help to check all the commands supported.:

```
IPMI_SIM>help
```

- Below tables show the detail information about each command.

Commands	Description
sensor info	Get all the sensor information.
sensor mode set <sensorID> <user>	Set the sensor mode to the user mode. Leaves the sensor reading as it currently is until instructed otherwise
sensor mode set <sensorID> <auto>	Set the sensor mode to the auto mode. Changes the sensor reading to a random value between the lnc and unc thresholds every 5 seconds.
sensor mode set <sensorID> <fault> <lnr | lc | lnc | unc | uc | unr >	Set the sensor mode to the fault mode. Changes the sensor reading to a random value to cause a particular type of fault as instructed (lnr, lc, lnc, unc, uc, unr) lower non-recoverable threshold lower critical threshold lower non-critical threshold upper non-critical threshold upper critical threshold upper non-recoverable threshold
sensor mode get <sensorID>	Get the current sensor mode.
sensor value set <sensorID> <value>	Set the value for a particular sensor..
sensor value get <sensorID>	Get the value of a particular sensor.
sel set <sensorID> <event_id> <'assert'/'deassert'>	Inject(Assert/Deassert) a sel error. You can use the sel set command to add a SEL entry for a particular sensor.
sel get <sensorID>	Get the sel error for a sensor. You can use the sel get command to get the available events for a particular sensor.

- Here's a example on how this console should be used and how it is chaning sensor readings. Let's prepare 2 terminal consoles: 1 for ipmi console and the other one is just normal console to use ipmitool to check how the manipulation works.

  1. First lets check processor temperature of virtual server:

```
sudo ipmitool -I lanplus -U admin -P admin -H localhost sensor get Temp_CPU0
    Locating sensor record...
    Sensor ID              : Temp_CPU0 (0xaa)
    Entity ID             : 65.1
    Sensor Type (Threshold)  : Temperature
    Sensor Reading        : 40 (+/- 0) degrees C
    Status                : ok
    Lower Non-Recoverable : na
    Lower Critical        : na
    Lower Non-Critical    : na
    Upper Non-Critical    : 89.000
```

```
        Upper Critical        : 90.000
        Upper Non-Recoverable : na
        Positive Hysteresis   : Unspecified
        Negative Hysteresis   : Unspecified
        Assertions Enabled    : unc+ ucr+
        Deassertions Enabled  : unc+ ucr+
```

2. Then let's peek and poke this sensor reading from 40 degree C to 85 degree C in ipmi console:

```
IPMI_SIM> sensor value get 0xaa
Temp_CPU0 : 40.000 degrees C
IPMI_SIM>
IPMI_SIM> sensor value set 0xaa 85
Temp_CPU0 : 85.000 degrees C
```

3. Last we can verify processor temerature sensor reading by issuing IPMI command again to check that sensor reading is really changed to 85 degree C:

```
sudo ipmitool -I lanplus -U admin -P admin -H localhost sensor get Temp_CPU0
    Locating sensor record...
    Sensor ID              : Temp_CPU0 (0xaa)
    Entity ID              : 65.1
    Sensor Type (Threshold)  : Temperature
    Sensor Reading         : 85 (+/- 0) degrees C
    Status                 : ok
    Lower Non-Recoverable : na
    Lower Critical         : na
    Lower Non-Critical     : na
    Upper Non-Critical     : 89.000
    Upper Critical         : 90.000
    Upper Non-Recoverable : na
    Positive Hysteresis   : Unspecified
    Negative Hysteresis   : Unspecified
    Assertions Enabled    : unc+ ucr+
    Deassertions Enabled  : unc+ ucr+
```

vSwitch Setup

You can implement the vSwitch component of InfraSIM by deploying the Cisco Nexus 1000v switch on the ESXi host.

For more information on downloading and using Cisco Nexus 1000v switch, refer to http://www.cisco.com/c/en/us/products/switches/nexus-1000v-switch-vmware-vsphere/index.html.

1.1.9 Contributing to InfraSIM

Contributions are welcomed and encouraged, in the form of issues and pull requests, but please read the guidelines in this section before you get involved.

Our project is relatively new, and we do not have many hard and fast rules. As the project grows and more people get involved, we will add to our guidelines, as needed.

Communicating with Other Users

We maintain a mailing list at https://groups.google.com/d/forum/infrasim. You can visit the group through the web page or subscribe directly by sending email to infrasim+subscribe@googlegroups.com.

We also have a #infrasim slack channel at https://codecommunity.slack.com/messages/infrasim/. You can receive an invite by requesting one at http://community.emccode.com.

Submitting Contributions

You can submit coding additions or changes for a repository. It's recommended that you limit your pull requests to a single issue, keep tests as simple as possible, and make sure your changes don't break the existing project.

1. Fork the repository and clone it locally.
2. Use a unique branch to make commits and send pull requests.
3. Make sure that the description of the pull request is clear and complete.
4. Run your changes against existing tests or, if necessary, create new ones.

After your pull request is received, our core committers give you feedback on your work and might request that you make further changes and resubmit the request. The core committers handle all merges.

If you have questions about the disposition of a request, feel free to email one of our core committers.

Core Committer Team

- Bryan.Fu@emc.com
- Robert.Xia@emc.com
- Mark.Ma@emc.com
- Forrest.Gu@emc.com

Please direct general conversation about how to use InfraSIM or discussion about improvements and features to our mailing list at infrasim@googlegroups.com

Reporting Issues

To report an issue or ask a question:

1. Go to https://github.com/infrasim/infrasim/issues.
2. Search the existing issues for your issue. Make sure your issue is not already reported.
3. If you have new information to share about an existing issue, add your information to the existing discussion.
4. **If you have a new issue, report it. Include the following information.**
    - Problem Description
    - Steps to Reproduce
    - Actual Results
    - Expected Results
    - Additional Information

Security Issues

If you discover a security issue, please report it in an email to Infrasim_core_committee@emc.com. Do not use the Issues section to describe a security issue.

Understanding the Repositories

The https://github.com/InfraSIM/InfraSIM repository acts as a single source location to help you get or build all the pieces to learn about, take advantage of, and contribute to InfraSIM.

Coding Guidelines

A best practice is to use the same coding style as the rest of the codebase. In general, write clean code and supply meaningful and comprehensive code comments.

Contributing to the Documentation

You can contribute to the InfraSIM documentation.

1. Clone the InfraSIM/docs repository.
2. Create a branch to make commits and send pull requests.
3. Make sure that the description of the pull request is clear and complete.

When your pull requests are merged, your changes are automatically published to the documentation site at http://infrasim.readthedocs.org/en/latest/.

Community Guidelines

Be respectful and polite to other community members. Make everyone in the community feels welcome.

1.1.10 Development Guide

Repositories

The InfraSIM repositories provide you with the code to set up, configure, and test a virtual environment consisting of simulated servers, storage devices, and smart PDUs. A thorough understanding of the individual repositories is essential for contributing to the project.

Application	Repository	Description
infrasim-compute	https://github.com/InfraSIM/infrasim-compute	infrasim-compute repository includes virtual BMC, and virtual host implementation. It simulates common functionalities of bare-metal servers and the properties and behaviors of servers from vendors like Kell, Quanta, etc. It re-implemented all virtual server features in a different way from what idic repo does. Major one is its package is application, instead of virtual machine template like what idic does.
IDIC	https://github.com/InfraSIM/idic	Legacy virtual compute implementation which packages virtual server node into one virtual machine template. Idic repository includes vBMC, vCompute, and vPDU. vBMC is the base OS of virtual BMC. vCompute simulates the common functionalities of a compute node and the behaviors of a generic server and several servers from vendors like Dell, Quanta, etc.
vpduserv	https://github.com/InfraSIM/vpduserv	Simulates the behaviors of the IPI PANDUIT PDU which conforms with vendor and open source specified licenses.
QEMU	https://github.com/InfraSIM/qemu	QEMU is a generic and open source machine emulator and virtualizer, more information please access http://wiki.qemu-project.org/.
OpenIPMI	https://github.com/InfraSIM/openipmi	OpenIPMI library, a library that makes it simple to build complex IPMI management software.
Test	https://github.com/InfraSIM/test	Scripts for InfraSIM automation and integration tests. It includes the test framework(puffer) and many test cases against the features InfraSIM provided.
Tools	https://github.com/InfraSIM/tools	Various tools and scripts to monitor and manage generic and common virtual nodes, virtual rack build.
vRacksystem	https://github.com/InfraSIM/vracksystem	The vRacksystem provides both REST APIs and WebGUI for deploying and configuring vNode/vPDU to compose virtual racks.
docs	https://github.com/InfraSIM/docs	The InfraSIM documentation available at http://InfraSIM.readthedocs.org/en/latest/.

Development conventions

- Guidelines for merging pull requests

For code changes, we currently use a guideline of lazy consensus with two positive reviews with at least one of those reviews being one of the core maintainers and no negative votes. And of course, the gates for the pull requests must pass as well (unit tests, functional test etc).

If you put a review up, please be explicit with a vote (+1, -1, or +/-0) so we can distinguish questions asking for information or background from reviews implying that the relevant change should not be merged. Likewise if you put up a change for review as a pull request, a -1 review comment isn't a reflection on you as a person, instead is a request to make a modification before that pull request should be merged.

- Pull request for a new feature is required to contain corresponding functional test.

3rd-party binaries notes

QEMU

InfraSIM leverages QEMU in its implementation. It introduced tested, stable major release from official QEMU repository. There are also additional code changes kept at infrasim/qemu for purpose of better simulating servers.

We always build QEMU on top of Ubuntu 64-bit 16.04 Linux and wrap it into one Debian package. This package is available at InfraSIM QEMU Debian. InfraSIM application will download and install it into system before starting its service.

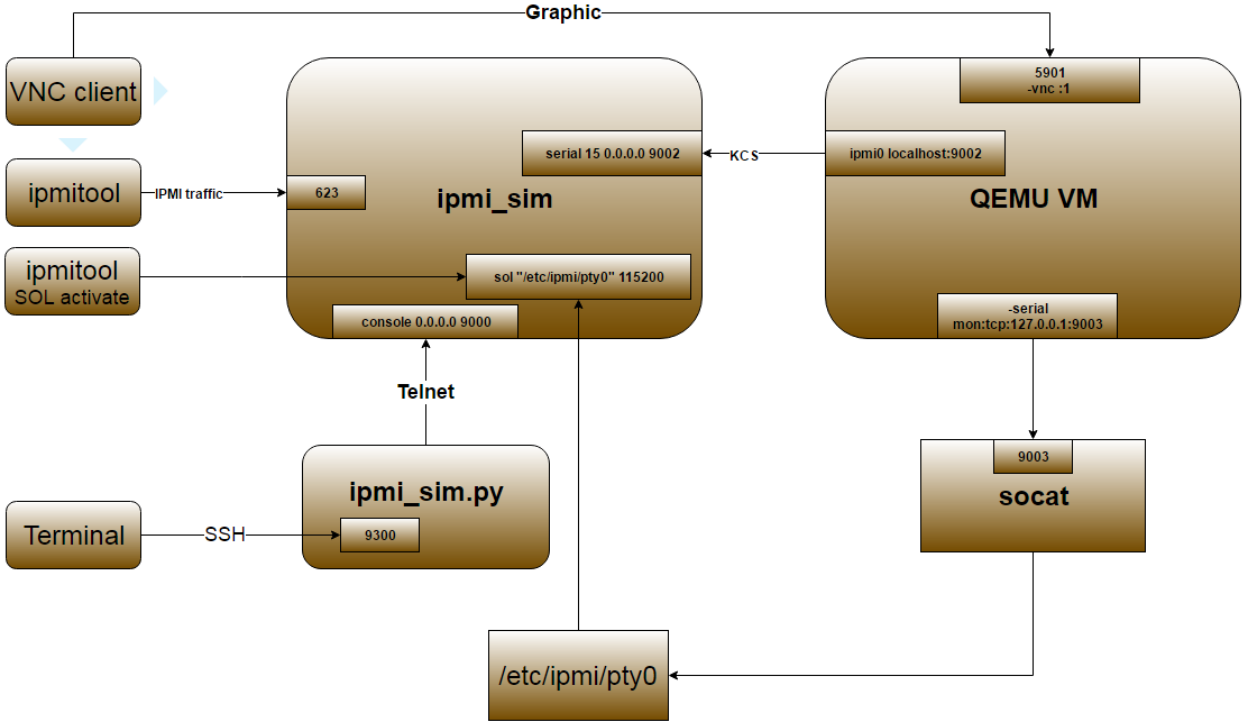
openipmi

InfraSIM leverages openipmi to simulate BMC properties and behavior. Similarly, there are also additional code changes kept at infrasim/openipmi for purpose of better simulating servers.

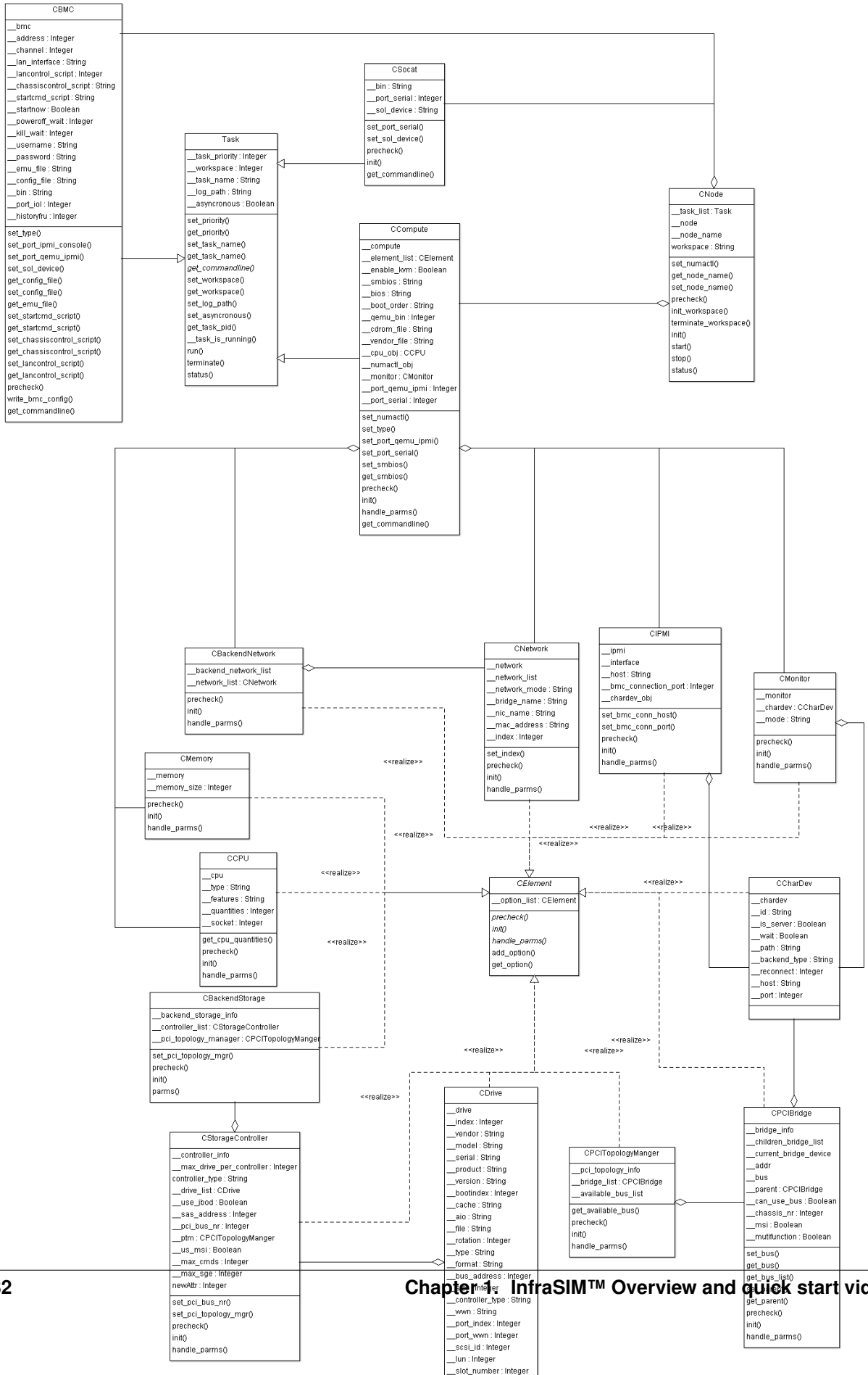
We always build openipmi on top of Ubuntu 64-bit 16.04 Linux and wrap it into one Debian package. This package is available at InfraSIM OpenIpmi Debian. InfraSIM application will download and install it into system before starting its service.

Component design notes

- infrasim-compute main components:
  1. Server node simulation
  2. IPMI consoles
  3. Server Emulation data
- Connection and communication path between modules:



- Class UML diagram of main components


CBMC
__bmc
__address : Integer
__channel : Integer
__lan_interface : String
__lancontrol_script : Integer
__chassiscontrol_script : String
__startcmd_script : String
__startnow : Boolean
__poweroff_wait : Integer
__kill_wait : Integer
__username : String
__password : String
__emu_file : String
__config_file : String
__bin : String
__port_iol : Integer
__historyfru : Integer
set_type()
set_port_ipmi_console()
set_port_qemu_ipmi()
set_sol_device()
get_config_file()
set_config_file()
get_emu_file()
set_startcmd_script()
get_startcmd_script()
set_chassiscontrol_script()
get_chassiscontrol_script()
set_lancontrol_script()
get_lancontrol_script()
precheck()
write_bmc_config()
get_commandline()
CSocat
__bin : String
__port_serial : Integer
__sol_device : String
set_port_serial()
set_sol_device()
precheck()
init()
get_commandline()
Task
__task_priority : Integer
__workspace : Integer
__task_name : String
__log_path : String
__asyncronous : Boolean
set_priority()
get_priority()
set_task_name()
get_task_name()
get_commandline()
set_workspace()
get_workspace()
set_log_path()
set_asyncronous()
get_task_pid()
__task_is_running()
run()
terminate()
status()
CNode
__task_list : Task
__node
__node_name
workspace : String
set_numactl()
get_node_name()
set_node_name()
precheck()
init_workspace()
terminate_workspace()
init()
start()
stop()
status()
CCompute
__compute
__element_list : CElement
__enable_lwm : Boolean
__smbios : String
__bios : String
__boot_order : String
__qemu_bin : Integer
__cdrom_file : String
__vendor_file : String
__cpu_obj : CCPU
__numactl_obj
__monitor : CMonitor
__port_qemu_ipmi : Integer
__port_serial : Integer
set_numactl()
set_type()
set_port_qemu_ipmi()
set_port_serial()
set_smbios()
get_smbios()
precheck()
init()
handle_parms()
get_commandline()
CBackendNetwork
__backend_network_list
__network_list : CNetwork
precheck()
init()
handle_parms()
CNetwork
__network
__network_list
__network_mode : String
__bridge_name : String
__nic_name : String
__mac_address : String
__index : Integer
set_index()
precheck()
init()
handle_parms()
CIPMI
__ipmi
__interface
__host : String
__bmc_connection_port : Integer
__chardev_obj
set_bmc_conn_host()
set_bmc_conn_port()
precheck()
init()
handle_parms()
CMonitor
__monitor
__chardev : CCharDev
__mode : String
precheck()
init()
handle_parms()
CMemory
__memory
__memory_size : Integer
precheck()
init()
handle_parms()
<<realize>>
<<realize>>
<<realize>>
<<realize>>
<<realize>>
CCPU
__cpu
__type : String
__features : String
__quantities : Integer
__socket : Integer
get_cpu_quantities()
precheck()
init()
handle_parms()
<<realize>>
<<realize>>
CElement
__option_list : CElement
precheck()
init()
handle_parms()
add_option()
get_option()
<<realize>>
CCharDev
__chardev
__id : String
__is_server : Boolean
__wait : Boolean
__path : String
__backend_type : String
__reconnect : Integer
__host : String
__port : Integer
CBackendStorage
__backend_storage_info
__controller_list : CStorageController
__pci_topology_manager : CPCITopologyManger
set_pci_topology_mgr()
precheck()
init()
parms()
<<realize>>
<<realize>>
<<realize>>
<<realize>>
CDrive
__drive
__index : Integer
__vendor : String
__model : String
__serial : String
__product : String
__version : String
__bootindex : Integer
__cache : String
__aio : String
__file : String
__rotation : Integer
__type : String
__format : String
__bus_address : Integer
__bus_number : Integer
__controller_type : String
__wwn : String
__port_index : Integer
__port_wwn : Integer
__scsi_id : Integer
__lun : Integer
__slot_number : Integer
CPCIBridge
__bridge_info
__children_bridge_list
__current_bridge_device
__addr
__bus
__parent : CPCIBridge
__can_use_bus : Boolean
__chassis_nr : Integer
__msi : Boolean
__mutifunction : Boolean
set_bus()
get_bus()
get_bus_list()
get_parent()
precheck()
init()
handle_parms()
CStorageController
__controller_info
__max_drive_per_controller : Integer
controller_type : String
__drive_list : CDrive
__use_jbod : Boolean
__sas_address : Integer
__pci_bus_nr : Integer
__ptm : CPCITopologyManger
__us_msi : Boolean
__max_cmds : Integer
__max_sge : Integer
newAttr : Integer
set_pci_bus_nr()
set_pci_topology_mgr()
precheck()
init()
handle_parms()
CPCITopologyManger
__pci_topology_info
__bridge_list : CPCIBridge
__available_bus_list
get_available_bus()
precheck()
init()
handle_parms()

Logging and debugging

Virtual serve application run-time log and error message are store at /var/log/infrasim/<node-name>/{openipmi.log, qemu.log}.

- "openipmi.log" logs the openipmi messages and errors.
- "qemu.log" logs the qemu messages and errors.

Other information need to check and is useful for trouble-shooting:

- InfraSIM virtual server run-time processes and argument list: socat, qemu and ipmi_sim

```
/usr/bin/socat pty,link=/root/.infrasim/node-0/.pty0,waitslave udp-listen:9003,reuseaddr

qemu-system-x86_64 -vnc :1 -name node-0-node -device sga --enable-kvm -smbios file=/root/.infras

/usr/local/bin/ipmi_sim -c /root/.infrasim/node-0/data/vbmc.conf -f /root/.infrasim/node-0/data/
```

- Check content of data file in runtime workspace. Refer to content in workspace

Unit test

Major programming language of InfraSIM is Python. Folder InfraSIM/test/unittest contains all Python unit test cases implementation http://pythontesting.net/framework/unittest/unittest-introduction/ explains what is Python unittest and guildelines of coming up test case.

Entry point of running unittest is InfraSIM/.unittests. Execute unit test by running:

```
cd infrasim-compute/
sudo ./.unittests
```

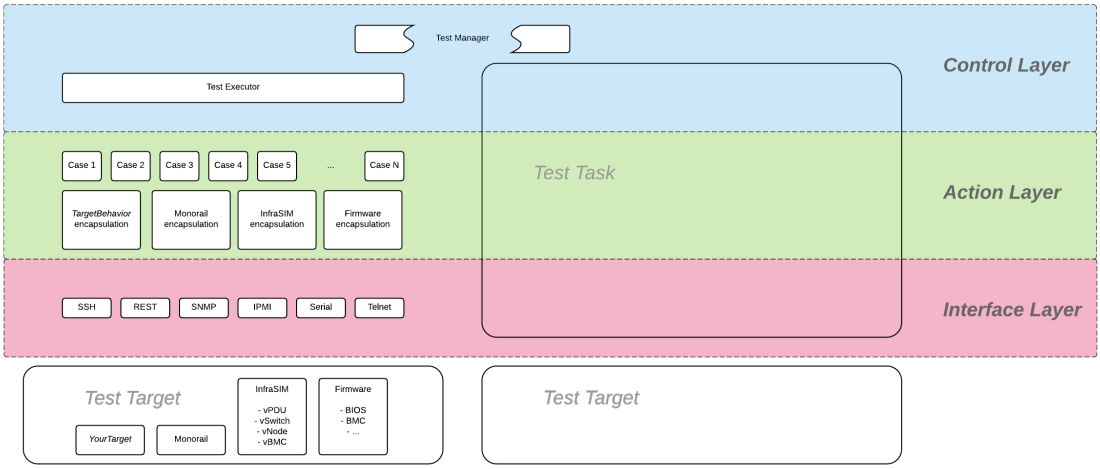
Functional test

Folder InfraSIM/test/functionaltest contains all the test cases to test virtual server implementation in functionality wise. Entry point of running functional test is InfraSIM/.functionaltests. Run below command to execute functional test:

```
cd infrasim-compute/
sudo ./.functionaltests
```

Integration test - under construction

Puffer is test framework developed for InfraSIM integration testing. Source code is in InfraSIM/test. It is a framework which can be easily extended to test products of different type, for example, standalone or web-based software and firmware. Here's its block diagram.



For any test target specified, those target behavior encapsulation need to be developed and a set of tests cases need to be added on top of encapsulation layer. Write test case described how to work out one test cases against InfraSIM. Below sections introduced all details about setting up buffer and execute InfraSIM testing with it.

Setup environment

Refer to the section 7.1 Physical Servers and ESXi Environment Setup.

Code:

```
git clone https://github.com/InfraSIM/test.git
```

Install necessary package:

```
sudo python test/install/PackageInstall.py
```

Define environment

You can see a configuration file example in test/configure/stack_example.json. To test your environment, you must define your environment in a file, and it must be in a valid JSON format.

1. Define the overall test environment.
   - *(Optional)* vRackSystem - The test may leverage vRackSystem and have REST talk.
   - available_Hypervisor - A list of hypervisors information. If your test has to handle hypervisors, this attribute is a required.
   - vRacks - A list of virtual racks you have built.

```
{
    "vRackSystem": {},
    "available_HyperVisor": [],
    "vRacks": [],
}
```

2. *(Optional)* Define vRackSystem key information for REST interaction, this definition can be an empty dictionary:

```
{
    "protocol": "http",
    "ip": "192.168.1.1",
    "port": 8888,
    "username": "admin",
    "password": "admin",
    "root": "/api/v1"
}
```

3. Specify hypervisor information using available_HyperVisor.

   For a single definition, here is an example:

```
{
    "name": "hyper1",
    "type": "ESXi",
    "ip": "192.168.1.2",
    "username": "username",
    "password": "password"
}
```

4. Specify a list of vRacks. Each definition includes:

   - name - any name you like.
   - hypervisor - The hypervisor you used in above definition. All virtual node, PDU, and switch are deployed on this hypervisor.
   - vPDU - A list of virtual PDU definition. The list can be empty.
   - vSwitch - A list of virtual switch definition. The list can be empty.
   - vNode - A list of virtual node definition. The list can be empty.

   They are organized in the following list:

```
{
    "name": "vRack1",
    "hypervisor": "hyper1",
    "vPDU": [],
    "vSwitch": [],
    "vNode": []
}
```

5. Specify a list of virtual PDUs. For each definition, you need to maintain:

   - name - virtual PDU's name in hypervisor
   - datatstore - on which datastore this PDU is deployed.
   - community - control community for SNMP access.
   - ip - PDU IP
   - outlet - A mapping of outlet to corresponding control password.

   Example:

```
{
    "name": "vpdu_1",
    "datastore": "Datastore01",
    "community": "foo",
    "ip": "172.31.128.1",
    "outlet": {
```

```
            "1.1": "bar",
            "1.2": "bar",
            "1.3": "bar"
        }
    }
```

6. vSwitch is currently not enabled.
7. Specify a list of virtual nodes. For each definition, you need to maintain:
   - name - The virtual node's name in hypervisor.
   - datastore - The datastore this node is deployed on.
   - power - A list of power control connection, each connection defines a specific PDU and outlet, you may have two power control, if this list is empty, node will not be controlled by any PDU.
   - network - A definition for connection to virtual switch, currently not used.
   - bmc - A definition on how to access virtual BMC of this node, including IP, username and password for ipmi over LAN access.

   Example:

```
    {
        "name": "vnode_a_20160126114700",
        "datastore": "Datastore01",
        "power": [
            {"vPDU": "vpdu_1", "outlet": "1.1"},
        ],
        "network": [],
        "bmc": {
            "ip": "172.31.128.2",
            "username": "admin",
            "password": "admin"
        }
    }
```

   Verify every IP is available from your test execution environment!

   Verify PDU can access substream hypervisor! (see chapter 7.1.3 vPDU Configuration for detail)

Case Runtime Data

Case Runtime Data used to maintain some specific data for different test objects. These data generally require the user to add and update manually. For example, if you want to test one type of sensor for multiple nodes, you need to add and update sensor ID corresponds to each node.

1. Configuration file:

   Case Runtime Data is defined in the json file which have same name with case script. If name of case script is T0000_test_HelloWorld.py, the name of runtime data shall be T0000_test_HelloWorld.json.

   Here's an example:

```
    [
        {
            "name_1": "value_1",
            "name_2": "value_2"
        }
    ]
```

If your configuration json like above, you can get "value_1" by call self.data["name_1"] in test case.

Here's another example:

```
[
    {
        "node_1": "0x00",
        "node_2": "0x01"
    },
    {
        "node_1": "0x02",
        "node_2": "0x03"
    }
]
```

If your configuration json has two objects in an array like above, same case shall be run twice for each runtime data.

You will get "0x00" by call self.data["node_1"] in test case for the first time, and "0x02" for the second time.

2. Test Result:

   You shall get two separate result and a summary. Case's final result is the worst result for all execution.

   For example, if the case "failed" in first time and "passed" in second time, the final result is still "failed", the summary will list all run results.

Run test

Trigger test:

```
cd test
python puffer.py -s infrasim --stack=<your_configuration>
```

<your_configuration> can be an absolute or related path of your configuration file. About how to run test, please check readme for detail:

```
cat README.md
```

You log file is kept in a folder of log/InfraSIM, each test task is packaged in a folder with time stamp as it's folder name.

Write test case

This section introduces how to write test case in puffer.

1. Create a test script file

   - **Test Case Name**

     The name of test case should follow the same format:

```
T\d+_[a-z0-9A-Z]+_[a-z0-9A-Z]+
```

     In puffer, test case name should:

     - Start with capital letter **T** and case id
     - Followed by the **field type** and **short description** about this case with underscores in the interval. Field types defined in class CBaseCase.

Note: The field type for InfraSIM is **idic**.

For example, a test case named T123456_idic_CheckPowerStatus:

- **T** is short for test
- **123456** for case id
- **idic** for field type
- **check the power status** for the short description

- **Test Suite**

  You should put your test case scripts into **<puffer_directory>/case/<test_suite>**. Each folder under **<puffer_directory>/case** is a test suite. When you give the suite folder to puffer.py as a parameter, puffer will executes all test case scripts which in the folder, including subfolders.

2. Create case runtime data file

   Case Runtime Data is used to maintain some specific data for different test objects. These data generally require the user to add and update manually.

   The format of case runtime data defined in the json file which have same name and folder with case script. Please see the chapter Case Runtime Data .

3. Write test case

   (a) Import CBaseCase

   Class CBaseCase defined in **<puffer_directory>/case/CBaseCase.py**, contains some member functions to help test case running:

```
from case.CBaseCase import *
```

   (b) Class Declaration

   We declaration each case as subclass of class CBaseCase and the class name is case name. For example, if case name is T123456_idic_CheckPowerStatus, the class name should be same to it.

   A test case maybe looks like:

```
from case.CBaseCase import *

class T000000_firmware_shortdescription(CBaseCase):

    def __init__(self):
        CBaseCase.__init__(self, self.__class__.__name__)

    def config(self):
        CBaseCase.config(self)

    def test(self):
        pass

    def deconfig(self):
        CBaseCase.deconfig(self)
```

   And then, we need to override methods of class CBaseCase, such as config(), test() and deconfig().

   (c) Override config()

   This method configuration system to expected status, configuration runtime HWIMO environment and stack environment.

The HWIMO configuration will set logger to save session log into log file and configuration SSH agent and stack configuration will build stack object, configuration stack ABS according to dict, build all nodes and power on.

However, in some case we want to enable some components we need to enable manually in configuration(). For example, if we want to use the ssh inside vbmc, we need enable the bmc_ssh in configuration():

```
def config(self):
    CBaseCase.config(self)
    self.enable_bmc_ssh()
```

(d) Override test()

This method is the main part of the test.

You can:

- Use self.stack to get the stack which build in config().
- Use self.data[] to get case runtime data.
- Use self.monorail to use Monorail API.
- Use self.log() to log the information.
- Use self.result() to save the case result.

For example:

```
def test(self):
    #get racks from stack and get nodes from rack
    for obj_rack in self.stack.get_rack_list():
        for obj_node in obj_rack.get_node_list():

            #log the information
            self.log('INFO', 'Check node {} of rack {} ...'
                .format(obj_node.get_name(), obj_rack.get_name()))

            #get and match outlet power
            for power_unit in obj_node.power:
                pdu_pwd = power_unit[0].get_outlet_password(power_unit[1])
                power_unit[0].match_outlet_password(power_unit[1], pdu_pwd)

            #virtual node power control
            obj_node.power_on()

            #use case runtime data
            node_name = obj_node.get_name()
            node_lan_channel = self.data[node_name]

            #send command to virtual bmc through ssh
            obj_bmc = obj_node.get_bmc()
            bmc_ssh = obj_bmc.ssh
            ssh_rsp = bmc_ssh.send_command_wait_string(
                str_command = 'ipmitool -I lanplus -H localhost -U {} -P {} lan print {} {}'
                wait = '$',
                int_time_out = 3,
                b_with_buff = False)

            #send command to virtual bmc through ipmitool
            ret, ipmi_rsp = obj_node.get_bmc().ipmi.ipmitool_standard_cmd('lan print')
```

```
                    #if case failed
                    if ret != 0:
                        self.result(FAIL, 'FAIL_INFORMATION')
                    else:
                    #if no issue in this run, case pass.
                        self.log('INFO', 'PASSED.')
```

(e) Override deconfig()

This method deconfig system to expected status, reset REST and SSH sessions, deconfig stack and log handler:

```
        def deconfig(self):
            self.log('INFO', 'Deconfig')
            CBaseCase.deconfig(self)
```

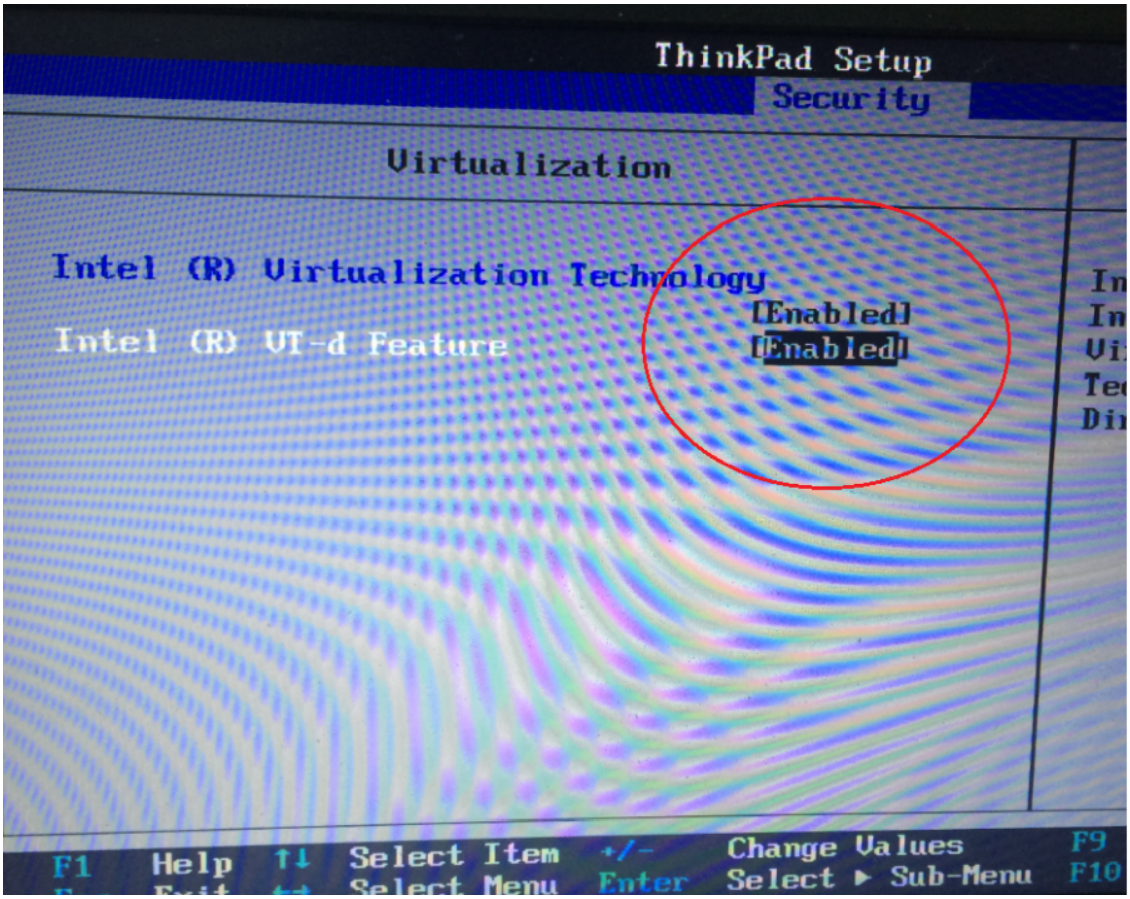
1.1.11 How To

How to install VMWare ESXi on Physical Server

1. **Requirement of physical server** The physical server must support ESXi 6.0 and it should be allocated at least 3 NIC ports. The first NIC port is used for the admin network connection. The second and third NIC ports are used for control network connection(The second NIC is required. The third NIC is optional). The fourth NIC port is used for data network connection (optional).

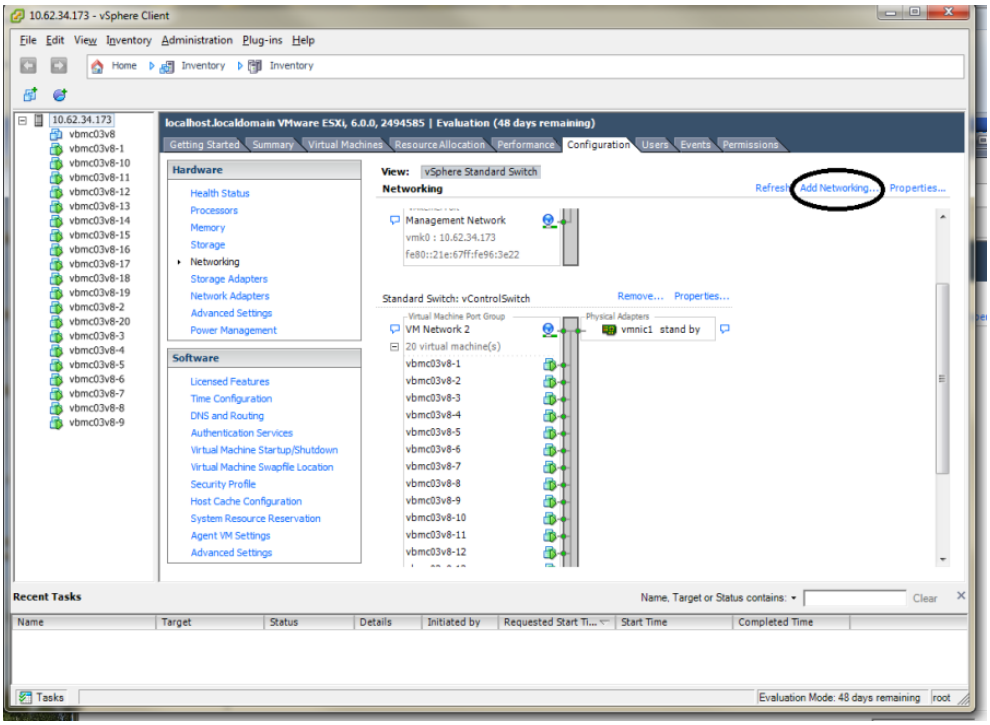
   Virtual InfraSIM servers runs in the best performance if hardware-assisting technology has been enabled on underlying physical machines. These technology includes VT-d feature and AMD-V for processors from Intel and AMD.

   Note: **Physical machine** - enable VT-d in BIOS

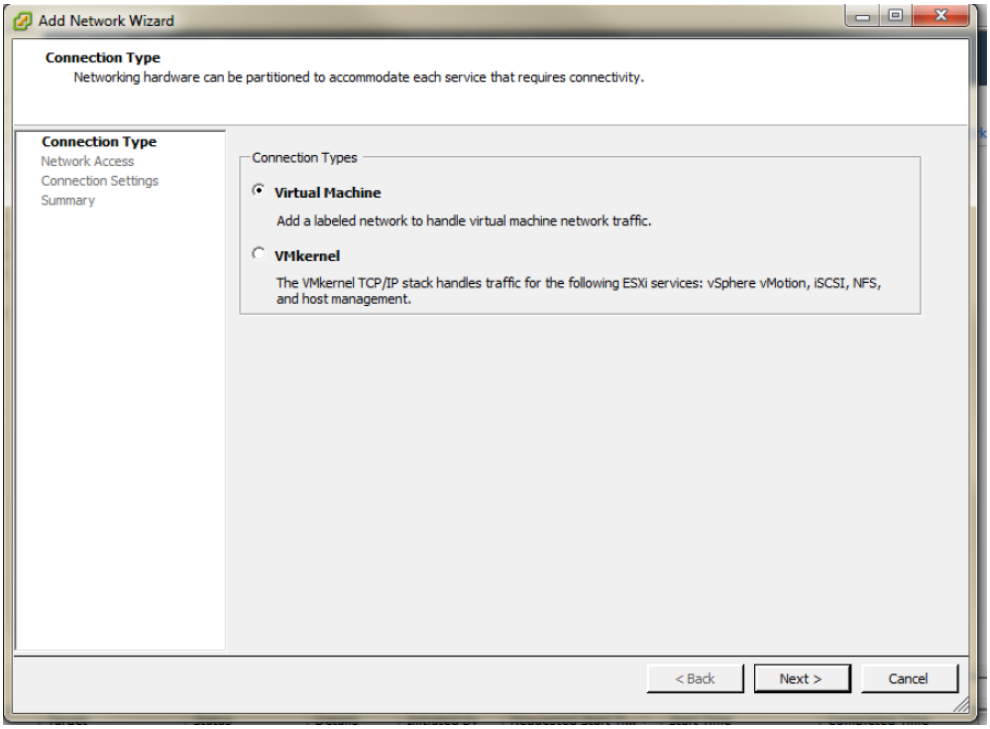


2. **Setting Up Network Connections** You must have IP addresses for the physical servers in the test environment to be used to configure the VMKernal port of ESXi and called as ESXi_Admin_IP.
    - Allocate or reserve a static IP address from the Lab admin.
    - Connect the server's admin NIC ports into the Lab network.
    - To set up a multiple server environment, connect Port C1 on each server by using an Ethernet switch.
3. **Install ESXi 6.0** From the VMWare web site, a 60-day free trial version is available after user registration.
    - Go to https://my.vmware.com/web/vmware/details?downloadGroup=ESXI600&productId=490&rPId=7539
    - Download the VMWare vSphere Hypervisor 6.0 (ESXi6.0) ISO image.
    - Install ESXi 6.0 on each physical server.
    - Configure the static IP address ESXi_Admin_IP on first NIC port.
    - Set the Administrator user name by using the format <User Name>.
    - Set the Administrator Password by using the format <Password>.
4. **Installing VMWare vSphere Client (Remote System)**
    - Go to the VMWare web site.
    - Download the VMWare vSphere Client.
    - Install the client on a remote system that can connect to the physical servers.
5. **Configuring the Virtual Network**

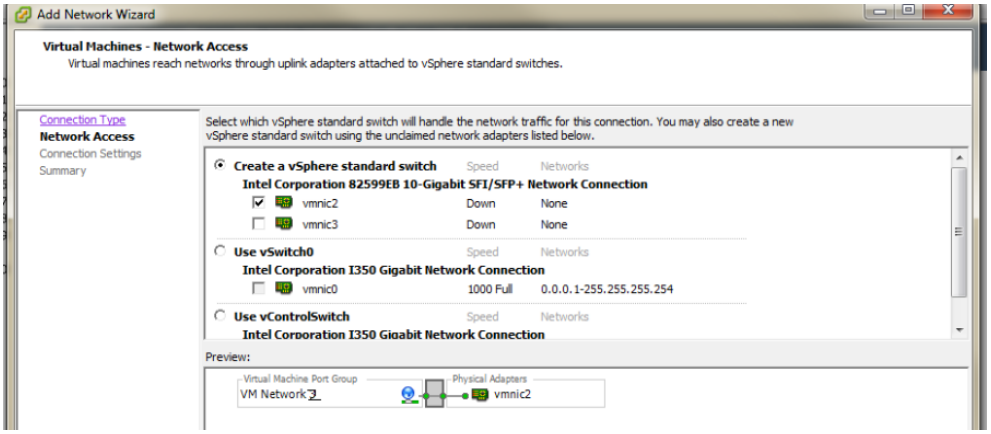
- Launch the vSphere client and connect to ESXi on the physical server by using ESXi_Admin_IP.
- On the Configuration tab, click Add Networking, to create the Control vSwitch. In the example, the network label is “VM Network 2”.



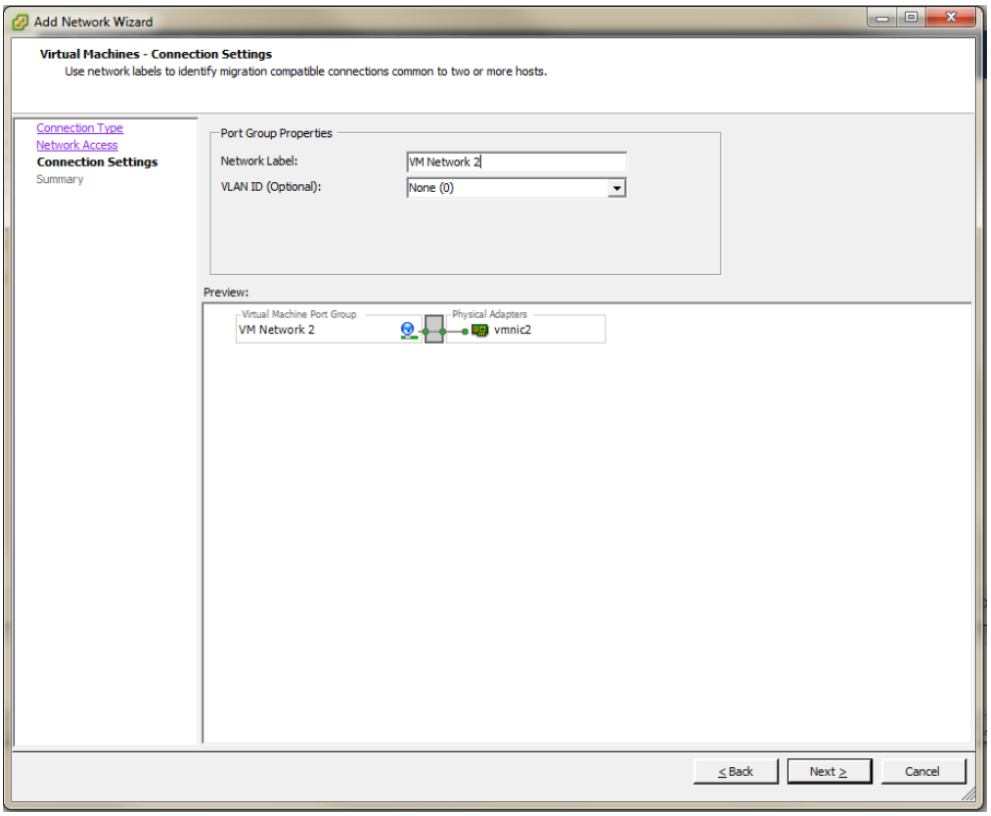
- Select Virtual Machine



- Select Create a vSphere standard switch > vmnic2.

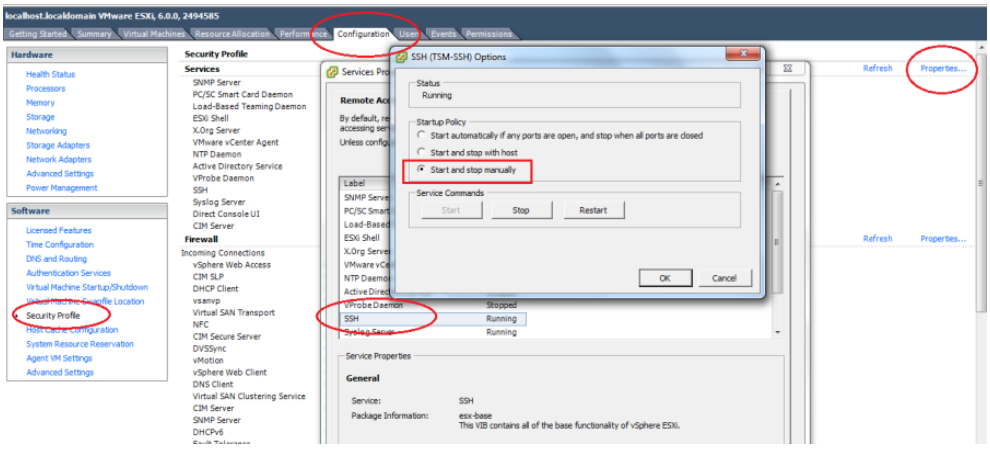


- In the Network Label field, type port group name on target switch.

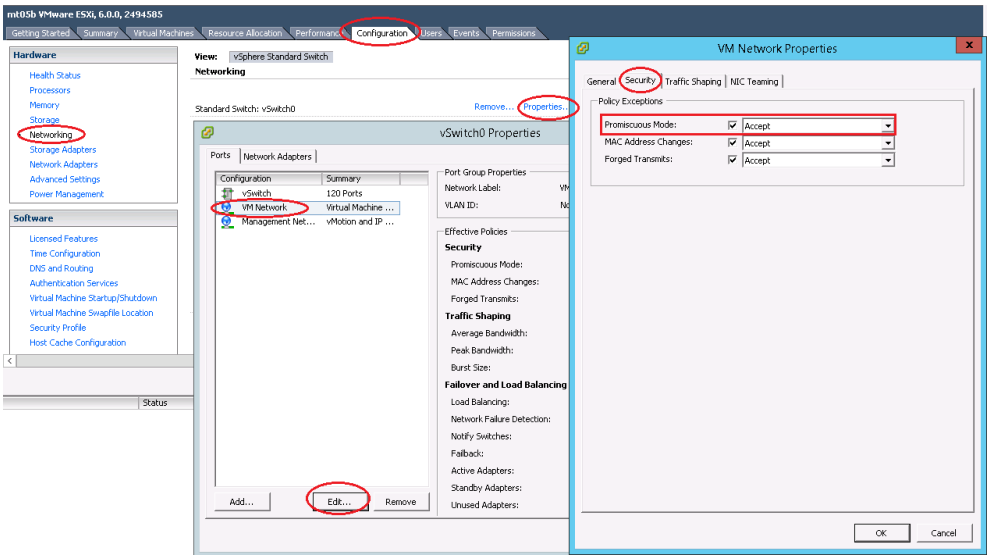


- Enable the SSH service on ESXi. To do this, open the Configuration tab and select Security Profile. Then select SSH and click Properties to set the SSH (TSM-SSH) to start and stop manually.

Note: Login to the ESXi server through SSH and echo by issuing the **"vhv.enable = "TRUE""** command to the /etc/vmware/config file. This command enables nested ESXi and other hypervisors in vSphere 5.1 or higher version. This step only needs to be done once by using the command: echo 'vhv.enable = "TRUE"' >> /etc/vmware/config.



Note: Set **Promiscuous Mode** to Accept and tick Override. To do this, open the Configuration tab and select Networking. Then click Properties of the vSwitch, choose port group, edit, security, tick the checkbox to override setting and select Accept.



How to deploy InfraSIM virtual server on different type of platforms

There are desires to deploy virtual server on different types of hypervisor like:

- VirtualBox
- KVM
- VMWare product, both VMWare vSphere or VMWare workstation

2 possible ways to achieve this:

- Create virtual machine image for corresponding hypervisor beforehand and them import that image onto hypervisors - InfraSIM application is ready in operating system running in virtual machines or containers on top of specified hypervisor or platform. These images are: OVA file for VMWare workstation or vSphere; QCOW2 file for KVM/QEMU; BOX or vagrant/VirtualBox, etc. Below listed some steps on how to deploy these template into different systems:

- Spin-up virtual machines running Ubuntu 64-bit 16.04 OS on desired hypervisor and then install infrasim-compute application. You may also leverage Chef or Ansible to deploy multiple virtual server instances into multiple virtual machines.

How to simulate another server - Under construction

InfraSIM also provided many utilities, interfaces for developers to build one simulation solution for a physical node that has not been supported by infraSIM. This sections walk through steps required to build one simulation for one specific server node.

1. To simulate a real hardware server, you have to get the server fru' data:

```
$ cd data
```

Under this directory, you can find "vnode.emu" file. In this file, we keep server fru' data here, like:

```
$ mc_add_fru_data 0x20 0x0 0x100 data \
  0x01 0x00 0x01 0x04 0x0f 0x00 0x00 0xeb \
  0x01 0x03 0x17 0x00 0xcd 0x51 0x54 0x46 \
  0x43 0x4a 0x30 0x35 0x31 0x36 0x30 0x31 \
  ......
```

You can use ipmitool to get BMC sensor's data:

```
$ ipmitool -U <your-account> -P <your-password> -I lanplus -H <your-BMC-IP> fru read <fru ID> fr
```

call fru_gen.py script to dump fru.bin to hex format:

```
$ cp ../../tools/data_generater/fru_gen.py ./
$ python fru_gen.py fru.bin
```

fru_result will be generated, replace original fru data with the expected one in this file.

2. Same as fru, in "vnode.emu" file, we keep server sensors' data here, like:

```
$ sensor_add 0x20 0x0 0x01 0x02 0x01
  main_sdr_add 0x20 \
  0x00 0x00 0x51 0x02 0x2a \
  0x20 0x00 0x01 0x15 0x01 0x67 0x40 0x09 0x6f 0x71 0x00 0x71 0x00 0x71 0x00 0xc0 \
  0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xcf 0x50 0x77 0x72 0x20 0x55 \
  0x6e 0x69 0x74 0x20 0x53 0x74 0x61 0x74 0x75 0x73
$ sensor_set_value 0x20 0x0 0x01 0x0 0x1

You can use ipmitool to get BMC sensor's data::

$ ipmitool -U <your-account> -P <your-password> -I lanplus -H <your-BMC-IP> sdr dump sensors
```

The above command will dump your server BMC sensors' data to the file named: "sensors" Generally, the sensor file contains binary data, we have to convert it to strings:

```
$ cp ../../tools/data_generater/sensors_gen.sh ./
$ ./sensor_gen.sh
```

After the command, you will get the file named: "all_sdr_sensors":

```
Use "all_sdr_sensors" file content to replace "vnode.emu" file of all "sensor_add" sections
    **Notice: This step is not necessary for your node unless you want to emulate the real BMC s
```

3. SMBIOS data is also needed, which can be got by using the command:

```
$ dmidecode --dump-bin <your-vnode-name>_smbios.bin
```

4. Build your vnode with real hardware fru, sensors and smbios data.:

```
$ make <your-vnode-name>
```

5. Enjoy your customized node.

How to simulate another vPDU - Under construction

InfraSIM provided ServerTech and Panduit PDU simulation initially. InfraSIM also provided many utilities, interfaces for developers to build simulation solution for other physical PDUs. This sections walk through all steps required to build one simulation for other PDU infraSIM doesn't support yet.

1. How to retrieve data from physical PDU

   If you want to retrieve PDU MIB data, you should have snmpsim installed on your environment.Then run the following command to produce MIB snapshot for the PDU:

```
# snmprec.py --agent-udpv4-endpoint=<PDU IP address>; --start-oid=1.3.6 --output-file=/path/<tar
```

   For more details of how to use snmprec.py, please go to section Producing SNMP snapshots at snmpsim home page for more help.

2. How to simulate physical PDU in InfraSIM

   Once you retrieved data from physical PDU, the next step is to add a virtual PDU in InfraSIM for this physical server. The following steps will guide you how to do:

   (a) Create a directory named **PDU name** at idic/vpdu

   (b) Create a directory data at idic/vpdu/<PDU name>/data, and copy the data you get from physical server into data directory.

   (c) Copy .config and Makefile into idic/vpdu/<PDU name>, and update target name in Makefile and .config

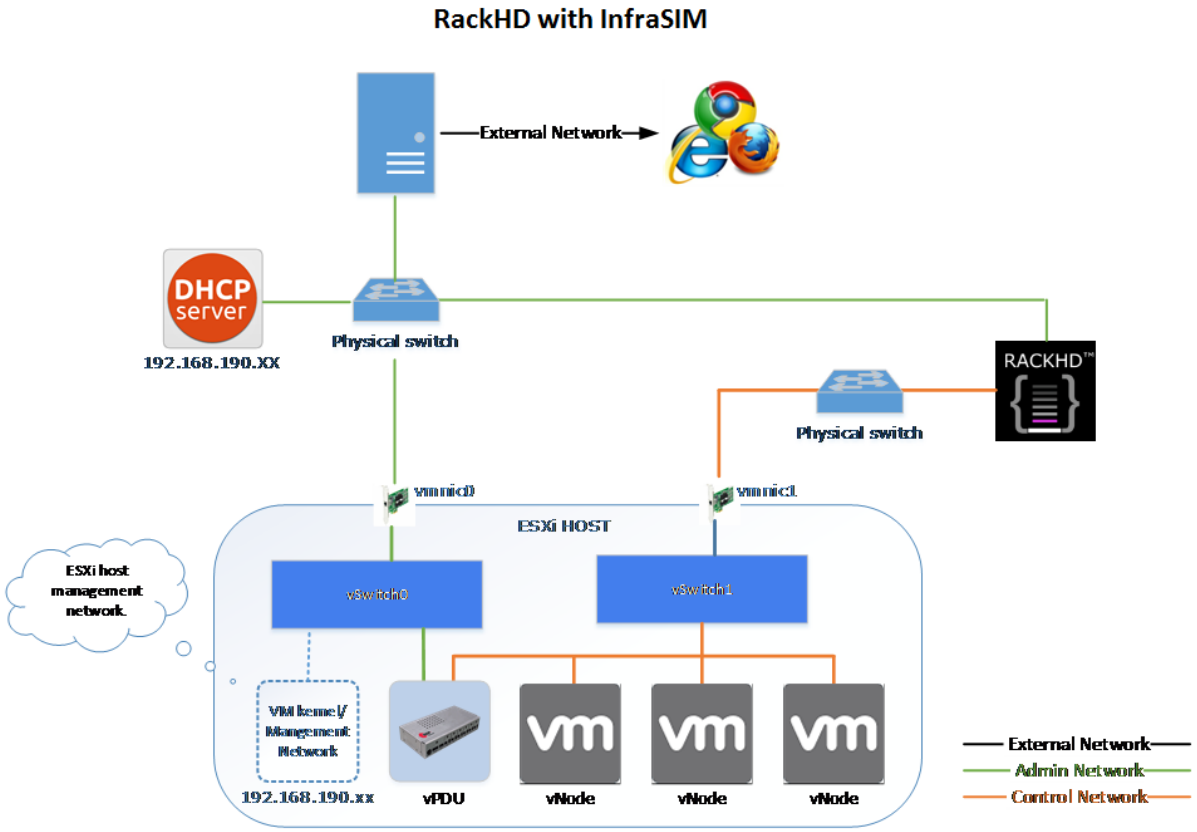
   (d) Clone vpduserv, and implement the new pdu logic based on vendor's PDU spec.

How to integrate RackHD with InfraSIM

RackHD is an open source project that provides hardware orchestration and management through APIs. For more information about RackHD, go to http://rackhd.readthedocs.io.

The virtual hardware elements(virtual compute node, virtual PDU, virtual Switch) simulated by InfraSIM can be managed by RackHD.

The following picture shows the deployment model for the integration of InfraSIM and RackHD:



Please follow up below steps to step the entire environment. After that, RackHD can discover and manage the virtual server and virtual PDU just as the real physical server and PDU.

1. **Please refer RackHD document(http://rackhd.readthedocs.org/en/latest/getting_started.html) to setup the RackHD Server**

   - "Admin Network" is used to communicate with external servers
   - "Control Network" is used to control the virtual servers.

2. Deploy virtual compute node and virtual PDU on ESXi with the network connection as in the pictures show. The ESXi show have network connection with the RackHD server.

After you setup the environment successfully, you can get the server information and control the servers by RackHD APIs. More information about how RackHD APIs communicate with the compute server and PDU, Please refer http://rackhd.readthedocs.org/en/latest/rackhd/index.html#rackhd-api

InfraSIM is a Trademark of EMC Corporation.