LINUX VOICE TUTORIAL

CONTROL VIRTUAL MACHINES WITH PYTHON AND LIBVIRT

VALENTINE SINITSYN

Learn ways to automate VM management when GUIs and simple shell scripts aren't enough.

WHY DO THIS?

- Automate virtual machine maintenance and management processes.
- Batch-create virtual appliances for clouds, integration testing and so forth.
- Get to know the de-facto standard virtualisation toolkit for Linux.

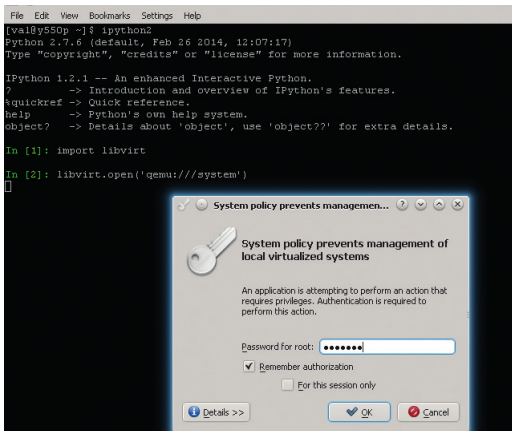
If you read Linux Voice, you are probably a Linux user. And if you use Linux, you most likely know what virtualisation is. Many mainstream distributions include KVM and virt-manager these days, and you can easily install Oracle VM VirtualBox, Xen or such like. Usually, they provide some form of GUI, so why on the Earth would you want to try virtualisation from a Python script?

If you just want to try out a new distro, you probably wouldn't. However, if you use several virtual machine managers (VMMs, or hypervisors) in parallel, or create pre-configured virtual machine appliances (say, for a cloud deployment), Python may come in handy.

Meet libvirt

Born at Red Hat as an open-source project, libivrt has become an industrial-grade toolkit that provides a generic management layer on top of different hypervisors, using XML as a mediation language. It's been adopted by many Linux vendors (if you have virt-manager, you have libvirt) and has bindings for many programming languages, including Python (version 2 and, starting with libvirt-python 1.2.1, Python 3). Libvirt can create (or "define", in its parlance), run ("create") and destroy virtual machines (called "domains" here), provide them with storage, connect them to virtual networks that are protected by network filters, migrate them between nodes and do other smart things.

However, libvirt has no convenient tools to work with XML, so you'll need to know the format (described at libvirt's website, **www.libvirt.org**) and use **xml.etree** or similar. Let's see it in action. Install libvirt's Python bindings (usually called **python-libvirt** or alike) and open an interactive Python shell (**>>>** denotes prompts in the listings below). No root privileges are initially required, but you may be asked to obtain them when needed.



Depending on the settings, you may be asked to enter the root password to use a system connection.

```
$ python
>>> import libvirt
>>> conn = libvirt.openReadOnly('qemu:///system')
```

Here, we import the **libvirt** module and open a connection to the hypervisor specified by the URI (note the three slashes). In this tutorial we'll work with Qemu/KVM, which is probably the most 'native' VMM for libvirt. **/system** means we connect to a local system-level hypervisor instance. You may also use **qemu:///session** to connect to the local per-user Qemu instance, or **qemu+ssh://** for secure remote connections. We are not going to define new domains now, so the restricted read-only connection will suffice.

For starters, let's check what your host is capable of when it comes to the virtualisation:

```
>>> xml = conn.getCapabilities()
>>> print xml
<capabilities>
  <host>
    <uuid>20873631-dad7-dd11-885a-08606eda31ae</uuid>
    <cpu>
      <arch>x86_64</arch>
      <model>Westmere</model>
      <vendor>Intel</vendor>
      <topology sockets='1' cores='4' threads='1'/>
            <feature name='vmx'/>
            ...
</capabilities>
```

You see how the XML is used to describe the host's capabilities. Libvirt identifies objects (hosts, guests, networks etc) by UUIDs. My host is a 64-bit quad-core Intel Core i5 with hardware virtualisation (VMX) support. Your results will likely be different.

The XML is quite long (note the ellipsis). Here's how you can use **xml.etree** to get supported guest domain types and corresponding architectures from it:

```
>>> from xml.etree import ElementTree
>>> for guest in tree.findall('guest'):
...   arch = guest.find('arch').get('name')
...   domain_type = guest.find('arch/domain').get('type')
```

My stock Ubuntu 13.10 supports Qemu domains only. However, since Qemu is a generic emulator, I can virtualise almost anything including s390x or SPARC (albeit at a performance penalty). x86_64 and i686 are of course supported, too.

It's good to know that you can create a domain for any conceivable architecture, but how do you actually do it? First of all, you'll need some XML to describe the domain. For simple cases, it may look like this:

```
<?xml version="1.0"?>
<domain type='qemu'>
 <name>Linux-0.2</name>
 <uuid>ce1326f0-a9a0-11e3-a5e2-0800200c9a66</uuid>
 <memory>131072</memory>
 <currentMemory>131072</currentMemory>
 <vcpu>1</vcpu>
 <os>
  <type>hvm</type>
            <boot dev='hd'>

 </os>
 <devices>
  <disk type='file' device='disk'>
   <source file='/path/to/linux-0.2.img'/>
             <target dev='hda'>
  </disk>
  <interface type='network'>
   <source network='default'/>
  </interface>
  <graphics type='vnc' port='5900'/>
 </devices>
</domain>
```

Speak the domain language

Here, we create a Qemu/KVM (hvm) virtual machine with one CPU and 128MB of RAM. It has a hard disk at IDE primary master (**hda**), from which it boots (I've used the tiny Linux 0.2 image from the Qemu Testing page). It is connected to the "default" network (NAT-enabled 192.168.122.0/24 attached to virbr0 at the host side), and you can use VNC at port 5900/tcp to access its screen (try **vinagre localhost:5900** or similar). Note that the **<source file="..."/>** must contain an absolute path to the image, and the image format must be supported by the hypervisor. libvirt is not a tool to create disk images, however you can use pyparted, ubuntu-vm-builder or similar to automate this process with Python.

Domains in libvirt are either transient or persistent. The former exist only until the guest is stopped or the host is restarted. Persistent domains last forever and must be defined before start. A transient domain will do for now, but as we are going to create something, a read-only connection is no longer sufficient.

```
import libvirt
xml = """domain definition here"""
conn = libvirt.open('qemu:///system')
domain = conn.createXML(xml)
```

Yeah, that's all. However, if you try to execute this script, you may get this response:

```
libvirt: QEMU Driver error : internal error: Network 'default' is not active.
```

This is because the XML references the "default" network, which won't be active unless there are domains using it already running, or you have marked it as autostarted with **virsh net-autostart default** command. Insert the following code just before **conn.createXML()** call to start the network if it is not already active:

```
net = conn.networkLookupByName('default')
if not net.isActive():
  net.create()
```

First, we get an object representing the "default" network. libvirt can look up objects by names, UUID strings (**ce1326f0-a9a0-11e3-a5e2-0800200c9a66**) or UUID binary values (**UUID('ce1326f0-a9a0-11e3-a5e2-0800200c9a66').bytes**). Corresponding method names start with the object's type (except for domains) followed by **"LookupByName"**, **"LookupByUUIDString"** or **"LookupByUUID"**, respectively.

Network objects provide other methods you may find useful. For instance, you can mark a network as autostarted with **net.setAutostart(True)**. Or, you can get an XML definition for the network (or any other libivrt object) with **XMLDesc()**:

```
>> print net.XMLDesc()
<network>
 <name>default</name>
 <uuid>9d3c0912-6683-4128-86df-72f26847d9d3</uuid>
 ...
</network>
```

If we were going to create a persistent domain, we'd change **conn.createXML()** to:

```
domain = conn.defineXML(xml)
domain.create()
```

There and back again

libvirt is essentially a sophisticated translator from a high-level XML to low-level configurations specific to hypervisors. Sometimes you may want to see what libvirt generates from your definitions. You can do this with:

```
>>> print conn.domainXMLToNative('qemu-argv', xml)
LC_ALL=C PATH=... QEMU_AUDIO_DRV=none /usr/bin/qemu-system-x86_64 -name Linux-0.2 ... -m 128 ... -smp 1,sockets=1,cores=1,threads=1 -uuid ce1326f0-a9a0-11e3- a5e2-0800200c9a66 ... -vnc 127.0.0.1:0 -vga cirrus...
```

Other times, you may be unsure how to express some VM configuration in XML, or you may have the configuration autogenerated by another front-end. libvirt can convert a native domain configuration to the XML with:

```
>>> argv="LC_ALL=C PATH=... QEMU_AUDIO_DRV=none /usr/bin/qemu-system-x86_64 -name Linux-0.2 ... -m 128 ... -smp 1,sockets=1,cores=1,threads=1 -uuid ce1326f0-a9a0-11e3-a5e2-0800200c9a66..."
>>> print conn.domXMLFromNative('qemu-argv', argv)

<domain type='qemu' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
 <name>Linux-0.2</name>
 <uuid>ce1326f0-a9a0-11e3-a5e2-0800200c9a66</uuid>
 <memory unit='KiB'>131072</memory>
 <currentMemory unit='KiB'>131072</currentMemory>
 <vcpu placement='static'>1</vcpu>
 <os>
  <type arch='x86_64' machine='pc'>hvm</type>
 </os>
 ...
</domain>
```

You can also use **virsh domxml-to-native** and **virsh domxml-from-native** commands for the same purposes.

(remember that persistent domain creation is a two-phase process). To gracefully reboot or shutdown the domain, use **domain.reboot()** and **domain.shutdown()**, respectively. However, the guest can ignore these requests. **domain.reset()** and **domain.destroy()** do the same, albeit without guest OS interaction. When the domain is no longer needed, you can remove (undefine) it like this:

```
try:
 domain = conn.lookupByUUIDString('ce1326f0-a9a0-11e3-a5e2-0800200c9a66')
 domain.undefine()
except libvirt.libvirtError:
 print 'Domain not found'
```

lookup*() throws **libvirtError** if no object was found; many libvirt functions do the same. If the domain is running, **undefine()** will not remove it immediately. Instead, it will make the domain transient. It is an error to undefine a transient domain.

When you are done interacting with the hypervisor, don't forget to close the connection with **conn.close()**. Connections are reference-counted, so they aren't really closed until the last client releases them.

Get'em all

A libvirt system may have many domains defined, and there are several ways to enumerate them. First, **conn.listDomainsID()** returns integer identifiers for the domains currently running on a libvirt system (unlike UUID, these IDs aren't persisted between restarts):

```
for id in conn.listDomainsID():
 domain = conn.lookupByID(id)
 ...
```

If you need all domains regardless of state, use the **conn.listAllDomains()** method. The following code mimics the behaviour of the **virsh list --all** command:

```
print ' Id    Name                           State'
print '-' * 52
for dom in conn.listAllDomains():
 print "%3s   %-31s%s" %\
   (dom.ID() if dom.ID() > 0 else '-',
               dom.name(),
               state_to_string(dom.state()))
```

For domains that aren't running, **dom.ID()** returns -1. **dom.state()** yields a two-element list: **state[0]** is a current state (one of **libvirt.VIR_DOMAIN_* constants**), and **state[1]** is the reason why the VM has moved to this state. Reason codes are defined per-state (see **virDomain*Reason enum** in the C API reference for the symbolic constant names). The custom **state_to_string()** function (not shown here) returns a string representation of the code.

Domain objects provide a set of ***stats()** methods to obtain various statistics:

```
cpu_stats = dom.getCPUStats(False)
for (i, cpu) in enumerate(cpu_stats):
 print 'CPU #%d Time: %.2lf sec' % (i, cpu['cpu_time'] / 1000000000.)
```

This way, you get a CPU usage for the domain (in nanoseconds). My host has four CPUs, so there are four entries in the **cpu_stats array. dom**. **getCPUStats(True)** aggregates the statistics for all CPUs on the host:

```
>>> print dom.getCPUStats(True)
[{'cpu_time': 10208067024L, 'system_time': 1760000000L, 'user_time': 5830000000L}]
```

Disk usage statistics are provided by the **dom.blockStats()** method:

```
rd_req, rd_bytes, wr_req, wr_bytes, err = dom.blockStats('/path/to/linux-0.2.img')
```

The returned tuple contains the number of read (write) requests issued, and the actual number of bytes transferred. A block device is specified by the image file path or the device bus name set by the devices/disk/target[@dev] element in the domain XML.

To get the network statistics, you'll need the name of the host interface that the domain is connected to (usually vnetX). To find it, retrieve the domain XML description (libvirt modifies it at the runtime). Then, look for devices/interface/target[@dev] element(s):

```
tree = ElementTree.fromstring(dom.XMLDesc())
iface = tree.find('devices/interface/target').get('dev')
rx_bytes, rx_packets, rx_err, rx_drop, tx_bytes, tx_packets, tx_err, tx_drop = dom.interfaceStats(iface)
```

The **dom.interfaceStats()** method returns the number of bytes (packets) received (transmitted), and the number of reception/transmission errors.

A thousand words' worth

Imagine you are making a step-by-step guide for an OS installation process. You'll probably do it in the virtual machine, taking the screenshots periodically. At the end of the day you will have a pack of screenshots that you'll need to crop to remove VM window borders. Also, it's pretty boring to have to sit there pressing PrtSc. Luckily, there is a better way.

libvirt provides a means to take a snap of what is currently on the domain's screen. The format of the image is hypervisor-specific (for Qemu, it's PPM), however, you can use the Python Imaging Library (PIL) to convert it to anything you want. To transfer image data from the VM, you'll need an object called **stream**. This provides a generic way to exchange data with libvirt, and is implemented by the **virStream** class. Streams are created with the **conn.newStream()** factory function, and they provide **recv()** and **send()**

Your mileage may vary

You may expect libvirt to abstract all hypervisor details from you. It does not. The API is generic enough, but there are nuances. First, you'll need your guest images in a hypervisor-supported format (use **qemu-img(1)** to convert them). Second, hypervisors vary in their support level. Qemu/KVM and Xen are arguably the best supported options, but we had some issues (like version mismatch or inability to create a transient domain) with libvirt-managed VirtualBox on our Arch Linux and Ubuntu boxes.

The bottom line: libvirt is great, but don't think you can change the hypervisor transparently.

methods to receive and send data. To get a stream containing the screenshot, use:

```
stream = conn.newStream()
dom = conn.lookupByUUID(UUID('ce1326f0-a9a0-11e3-a5e2-0800200c9a66').bytes)
if dom.isActive():
 dom.screenshot(stream, 0)
```

Here, we lookup the domain by a binary UUID value, not a string (the UUID class comes from the **uuid** module). We check that the domain is active (otherwise it has no screen) and ignore other possible errors. Now we need to pump the data to the Python side. **virStream** provides a shortcut method for this purpose:

```
buffer = StringIO()
stream.recvAll(writer, buffer)
stream.finish()
```

Here, we create a StringIO file-like object to store image data. **stream.recvAll()** is a convenience wrapper that reads all data available in the stream. **writer()** function is defined as:

```
def writer(stream, data, buffer):
 buffer.write(data)
```

Its third argument is the same as the second argument in **recvAll()**. It can be an arbitrary value, and here we use it to pass the **StringIO()** buffer object.

All that remains is to save the screenshot in a convenient format, like PNG:

```
from PIL import Image
buffer.seek(0)
image = Image.open(buffer)
image.save('screenshot.png')
```

PIL is clever enough to autodetect the source image type. However, it expects to see the image data from byte one, that's why we use **buffer.seek(0)**.

You can easily wrap this screenshotting code into a function and call it periodically, or when something interesting happens to the VM.

You've got a message

When something happens to a domain, for example it is defined, created, destroyed, rebooted or crashed, libvirt generates an event that you can subscribe to and act appropriately. To be able to receive these events, you'll need some event loop in your code. libvirt provides a default one, built on top of the blocking poll(2) system call. However, you can easily integrate with Tornado IOLoop (LV1) or glib MainLoop (LV2), if needed.

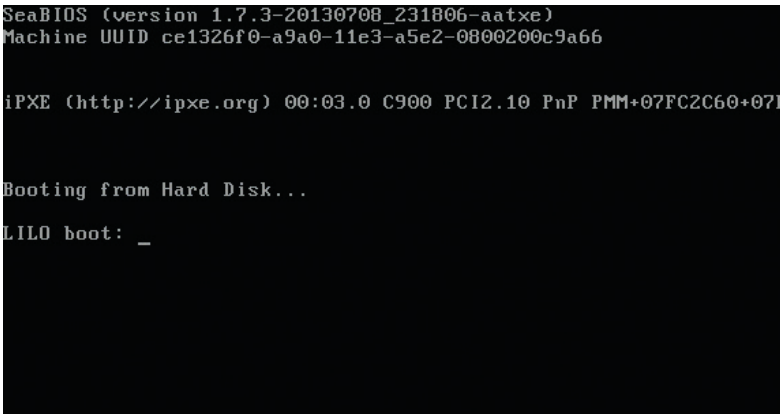
Default event loop is registered at the very beginning, even before the connection to libvirt daemon is opened:

```
libvirt.virEventRegisterDefaultImpl()
conn = libvirt.open('qemu:///system')
```

Next, you subscribe to the events you are interested in. Let's say we want to receive events of any type:

```
cb_id = conn.domainEventRegisterAny(None, libvirt.VIR_DOMAIN_EVENT_ID_LIFECYCLE, event_callback, None)
```

The first argument is the domain we want to monitor; None means any. The second argument specifies the event "family" to subscribe to. Here, we are interested in lifecycle events (started, stopped, etc), but there are many others (removable device changed, power management occurs, watchdog fired, and so on). The last argument is an arbitrary value to be passed to the **event_callback()** function (remember **stream.recvAll()** and **writer()** we saw earlier?).



You can take a screenshot of the VM as early as you want, even before a guest kernel is booted.

Event handler is defined as follows:

```
def event_callback(conn, domain, event, detail, opaque):
 print 'Event #%d (detail #%d) occurred in %s' % (event, detail domain.name())
```

event and **detail** are integer codes describing what happened. For lifecycle events, they are defined in the **virDomainEventType** and **virDomainEvent*DetailType** enums; the constants (**libvirt.VIR_DOMAIN_EVENT_STARTED** etc) are named the same as enum fields.

```
while True:
 libvirt.virEventRunDefaultImpl()
```

This is the main loop. In a real application, you will probably run it in a separate thread. The call blocks until a subscribed event (or a timeout) occurs, so even exiting with Ctrl+C takes some time.

When the subscription is no longer needed, you can terminate it with:

```
conn.domainEventDeregisterAny(cb_id)
```

Events notification opens many interesting possibilities. For instance, you can start domains in the particular order (one after another), or use the Tornado framework to create a lightweight web-based virt-manager alternative.

And there's more...

This concludes our quick tour of the features of libvirt. We've barely scratched the surface, and there is much more than we've seen so far: storage pools, encryption, network filters, migrations, nodes, Open vSwitch integration and the rest. However, the APIs you've learned today form a solid foundation to build more advanced libvirt skills for your next project. Let the computer do the repetitive work for you, and have fun with Python in the meantime!

Dr Valentine Sinitsyn has committer rights in KDE but spends his time mastering virtualisation and doing clever things with Python.