VM plans

Initial Goal

..is to run Gleam in a VM. Probably that entails

1. Make an "appliance": rhel5 or rhel6 VM with Gleam executable and all required libraries, job options, etc. To start, would be simplest to run MC job, write output to VM local disk.
2. Write program or script for host machine which will do one of the following:
    - start up Gleam in the already-up-and-running VM
    - first boot up the VM, then ask it to run Gleam

Issues and choices

For the most part the following sections concern decisions for the longer term, not just for the initial goal of running Gleam in a VM, in fact some of them are irrelevant except in the longer term.

Which virtualization software?

I'm experimenting with VirtualBox to start, but since we're only concerned with Linux for now VMware (VMplayer product) would probably serve; there may be other suitable candidates, but apparently only a couple others are typically used to provide "service continuity", and they are proprietary. Advantages of VirtualBox compared to VMware are

- **Supported OSes**
    - VirtualBox supports Linux, Windows and Mac for client and host (though there are a bunch of restrictions for Mac, mostly due to the way Apple does things).
    - VMplayer supports Linux and Windows
- **API**
    - VirtualBox API (the same one used internally, hence complete) supports C++, python, and any other languages that understand (Microsoft) COM or, on non-Windows platforms, XPCOM for applications running on the host. There is also a web service which implements nearly the complete API. Any language with a toolkit for wsdl can use this. It's especially easy to use from Python and Java because the VirtualBox SDK includes wrapper classes for these languages.
    - The free SDK product VIX can be used with VMplayer. Functionality is somewhat limited compared to use with non-free products VMworkstation and vSphere, but may be sufficient for us. In particular, it does include the ability to start a program on a VM. Bindings exist only for C, Perl, and (Microsoft) COM.

Minimum set of functions to be provided on or for VMs

- Run batch (pipeline) jobs using Gleam, both for MC generation and for real data reconstruction
- Build Gleam and related software (need access to legacy compiler, external libs) e.g. via Release Manager
- Provide development environment for Gleam programmers (interactive login, editor, debugger, etc.).
- Allow for creation and installation of new version of VM including, for example, new build of Gleam

Interaction with lsf

How aware of VMs should lsf be? VMs could be routinely up and running, in which case lsf could treat them as part of its pool. Or a job submitted to lsf could be directed to run on a VirtualBox host. That job might then do some part of the scheduling an resource allocation normally handled by lsf. (In fact, if the webinterface were used, the lsf job need not even run on the VirtualBox host, but I don't see any advantage to this.)

What should a batch job do?

This will depend on the state of the VM before the job is run. From the VirtualBox API one could in principle create the VM, configure it, boot it and ask it to run something all from the API, but for our use we'll want to, at the very least, create, configure and register the VM - or perhaps just a template VM to be cloned on a per-job basis - beforehand. Booting a VM for every job start does of course entail some overhead (SL6 takes a couple minutes to boot up on my 2.26 GHz dual core laptop, running rhel5), but that may be tolerable in the context of our typical multi-hour batch job. Alternatively, if the (presumably SCCS-maintained) host machine is dedicated to our use, some number of guest VMs can be up and running at all times. In that case we'd want to monitor, to be sure they really *are* up and usable, and ideally have an automated procedure to reboot any crashed or hung VMs.

Resource access and security

For our intended uses VMs must have input from the outside world and must be able to write output somewhere which is accessible to non-VMs ~~however the VM is perhaps running an unpatched legacy OS with known security holes.~~ **Update:** Redhat has announced that rhel5 and rhel6 will have a 10-year production cycle (as opposed to rhel4 , which was only 7). rhel6 will be maintained as a production OS into November 2020, beyond the 10-year lifetime of Fermi.

There is no need for the VMs to be general purpose machines. Restrictions like the following should keep the machines and other SLAC resources from being trashed without interfering with the functions we require. (Even though, in light of 10-year rhel6 lifetime, these restrictions are no longer crucial, it's still probably a good idea to implement most if not all.)

- VMs should not have write access to any "regular" public space (space to which centrally-supported public log-in machines also have access).
- VMs need have no access at all to SLAC user home directories.
- Access to any device to which VM may write should be carefully controlled and limited to selected user ids. Output from VM jobs can be copied to a public area after vetting.

- Interactive login to VMs normally used for batch should be restricted to a small number of users (using usernames distinct from their regular SLAC usernames?), just those involved in maintenance of the VMs and perhaps some developers if there are needs that can't be met by the dedicated interactive login VMs
- Interactive login to VM development machines should also be restricted: to VM maintainers and active developers.
- VMs should only have software installed if it's required; everything else should be stripped out; in particular, no browsers and no email programs.
- Ports which are not required should be disabled.

However VMs will need some communication with the outside world, including at least:

- checkout from and commit to a CVS repository. It could be separate from the usual repository but it should be browsable from non-VM machines via ViewVC or similar product.
- read (mostly) and occasionally write to MySQL databases.
- VMs used for batch need to write output. Developers using VMs interactively need work space.
- Developers may need to run X applications on VMs, but cutting and pasting from or to such applications should perhaps be disallowed.

Configuration, performance, $$

Currently (according to BaBar experts who are much further along in their VM plans) multi-core VMs don't perform nearly as well relative to single-core machines as their hardware counterparts, so the better strategy is to instantiate many single-core machines. But each one needs a license. That many rhel6 licenses would be prohibitive, so they went to Scientific Linux instead. We may want to do the same. Or things may have changed by the time it matters to us.

Experiences with VirtualBox

Installation on my rhel5 laptop was straightforward, as was creating and starting up a VM using the gui application. The only hitch was that redhat4 - actually the Scientific Linux equivalent - did not make a good guest: it failed to boot. SL5 and SL6 guests are ok. The VirtualBox doc warns that kernels which are too old, including the one in rhel4, have problems. In fact, they also discourage use of 2.6.18 (rhel5 and SL5 kernel) in the guest, claiming it suffers from race conditions at boot, but I haven't seen any signs of that.

SDK

Part of the attraction of VirtualBox is the comprehensive API. It turns out this is not part of the standard install; the interface and some sample programs are in a separate platform-independent zip file. Instructions for installations were practically non-existent. Some essential steps (e.g., appropriately defining a couple environment variables) were not mentioned anywhere, but by reading the installation script, etc., I think I figured out what was required and now have a usable installation.

The next step was to try the C++ sample program which

1. Connects to the Virtual Box manager (starts it up if not already running)
2. Lists virtual machines which are already known to the Manager
3. Creates a new VM

There were some hard-coded file paths, etc., which caused 3. to fail. However that step still failed even after I fixed them (all the ones I'm aware of, anyway). At least it fails further along in the process - the VM is created and a (virtual) disk is created for it, but the operation of attaching the disk to the VM fails. Since I don't expect to use the API to create VMs (only to activate them and run programs on them), there is no reason to pursue this.

SDK cont'd

March 15th update:The immediate goal is to write a C++ program using the sdk which will start up a pre-defined VM and get it to run some executable (and then perhaps shut it down). So far I've managed to get through the start-up step. This was much more painful than expected but, now that I've learned something about COM, xpcom, and where to look among VirtualBox source for interface information - there is formal and reasonably complete documentation for the API, but not really geared for use from C++ - the last bit should be easier.

March 19th update: I was able to write a C++ program to accomplish the second step (run a program on the VM) but so far only after the VM was up and running. If I try to do it all in one program the second step fails, I assume because the VM is in some sense not ready.

June 11th update: Test program boots the VM, checks for "running state", checks for availability of disk and attempts to locate the particular script it intends to run. Even when all these checks succeed, the running of the script can still fail. However, it succeeds if a sleep of 100 seconds is inserted before attempting to run the script. 60 is not enough. This is not a very satisfactory state of affairs.

Other Interfaces

VirtualBox provides a variety of ways to get at the core functionality. Approximately from least to most grungy:

VBox gui (canned program). It exports nearly the complete API. I used it to create Scientific Linux 5 and Scientific Linux 6 VMs (after downloading ISO images from the SL web site) and use it routinely to monitor VM activity. It's fine for interactive use; not appropriate for running batch jobs on VM

VBoxManage (canned program). Command-line interface which exports complete API. It could conceivably be used to start batch executions. The command to start a program expects a plain text password; maybe it could be hidden in a script variable. I've looked over the source and exercised the program in order to gain insight into how things are supposed to work.

VirtualBox shell - extensible python interface. Have not yet tried it, but probably should. It's only barely mentioned in the SDK document, but some source is installed as a Python site package. The main program is (relative to installation directory)

sdk/bindings/glue/python/sample/vboxshell.py

On my machine that's /usr/lib/virtualbox/sdk/bindings/glue/python/sample/vboxshell.py

See SDKRef.pdf, section 2.1.2, for a little more information.

Web service - exports nearly complete API. Can be used via Java, Python, or as raw web service. I don't think it's appropriate for us. We don't need the ability to control VMs remotely; I expect VM commands will be issued from the host.

COM/XPCOM (COM is Component Object Module, Windows only; XPCOM is an open source implementation of COM used on non-Windows platforms). This is what the VBox gui and VBoxManage use internally, and it's the mechanism I've been exploring for us so far.