A Knoppix-based demonstration environment for JPF

Richard Potter
School of Information Science and Technology
The University of Tokyo, Tokyo, Japan
potter@is.s.u-tokyo.ac.jp

Cyrille Artho
Research Institute for Secure Systems
AIST, Amagasaki, Japan
c.artho@aist.go.jp

Kuniyasu Suzaki
Research Institute for Secure Systems
AIST, Tsukuba, Japan
k.suzaki@aist.go.jp

Masami Hagiya
School of Information Science and Technology
The University of Tokyo, Tokyo, Japan
hagiya@is.s.u-tokyo.ac.jp

ABSTRACT

This paper explores how a KVM virtual machine booted with a Knoppix Live DVD can provide a simple and reliable system for sharing demonstrations of JPF, and in particular, for running regression tests in a repeatable way before changes are committed to shared repositories. To make the system easy to automate, we integrated host file system access and a server for script execution. To make it practical for an interactive workflow, checkpointing was added to avoid booting and configuration delays. As an unexpected benefit, the isolation provided by the virtual machines allows multiple tests to run in parallel without risk of clashes over resources such as server ports.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*

General Terms

Experimentation

Keywords

Virtual Machines, Regression Testing, Demonstrations

1. INTRODUCTION

One of the first motivations for the development of virtual machines (VMs) back in the 1970's was the development and testing of software for legacy systems [4]. VMs are effective for this because they can be used to reproduce almost every detail of a system, including hardware interfaces, operating systems, and software libraries. With the recent trends towards distributed development, this same capability now enables software developers and researchers to confidently know that they share the exact same computer configuration, which makes comparison of results and overall collaboration more efficient [2, 6].

Figure 1: Starting from the same virtual machine software and repeating same steps gives reproducible demonstrations.

We became interested in virtual machines' ability to facilitate collaboration in JPF development because of inconsistent results when running regression tests. After confirming newly written code passed regression tests, a developer would commit code only to learn that it made regression tests fail on another developer's machine. For JPF [11] and net-iocache [1], numerous differences between developers' machines can affect results. Subtle configuration changes in JPF can be hard to keep track of, especially when a developer is working with multiple configurations. Net-iocache uses peer server applications, which may tie up network ports if a peer fails to shut down from a previous test. For debugging, developers might make changes in code or configuration and forget to undo some. Having all developers demonstrate their code passes regression tests on a standard test machine helps remove these differences.

Virtual machines provide a way to share exact copies of the same test machine with every developer. Each copy can be trusted to provide an equivalent environment because the VM builds it up from software components that can be verified to be exactly the same. If each developer then installs the same software and runs the same tests in the same way, the test results will likely be the same (Figure 1). Differences, if any, can be narrowed down to just a few causes mostly involving the test's sensitivity to subtle timing issues.

While it is clearly possible to use VMs as standardized test machines, we started to wonder whether such use could be fit into our workflow smoothly. Could each developer set it up and use it easily? Could we trust it? How hard would it be to develop a workable solution?

With easy development in mind, our first idea was simply to boot a VM using a Knoppix Live DVD image. Setup would be easy because all it required was downloading one Knoppix ISO file and booting it with KVM using a short command line. However, using it was not easy enough. It took quite a few manual steps to copy our project into to the VM and run the regression tests. Waiting for each step to complete added to the tedium. On the bright side though, Knoppix plus KVM did run the regression tests successfully. And it did so in a way that we could easily understand and trust, because it used only two open source components, KVM and Knoppix, which all developers could independently acquire and easily verify to be unchanged. These strengths made us want to find a way to make this combination easier to use.

Automating all the manual steps was the obvious way to improve ease of use and also to increase trust, since an automated solution would guarantee that steps were performed exactly the same. So the question became whether it was possible to provide automation in a way that preserved the way that KVM and Knoppix were easy to set up and easy to trust.

Our solution comes in two parts. The first part is a general script we call `dinkvm` [7], for *"do in Knoppix virtual machine"*, which automates the process of booting a fresh Knoppix virtual machine and then running a script inside. The second part is a script that automatically loads the latest commit directly from a developer's project directory, does a fresh build, and runs regression tests. `dinkvm` has additional features that make scripts easier to write and run. However, the core simplicity is retained so that, if necessary, anybody can run the script without `dinkvm`, using only standard KVM and Knoppix downloads.

2. BASIC USAGE

Dinkvm can be installed simply by cloning its repository. The cloned local copy can be put anywhere, and there is no compilation step. The one addition step that there is, is to download the Knoppix image from one of the mirrors listed at `http://www.knopper.net`. This 4 GB image file can simply be placed inside the clone of the repository directory.

Assuming a demonstration script has already been set up, a developer can invoke it like this:

```
$ cd /path/to/work/directory
$ /path/to/dinkvm -dofresh /path/to/script.sh
```

This will run the script in a virtual machine that has been freshly booted with a Knoppix DVD ISO image. The script runs inside the virtual machine as user knoppix, with the current directory set to `/home/knoppix`. The script has access to an X Windows display and can access any files on the host at `/path/to/work/directory` by using the path `/home/knoppix/onhost`. Output from the script is forwarded to dinkvm's standard output. When the script terminates, the virtual machine is removed.

The above example shows the two core parameters: the script to run and the host directory to make available to the virtual machine. Other parameters exist, but can usually be ignored because they receive default values that are usually acceptable. No root access privileges are required, so any user can run these commands.

For a user who wishes to package and share a reproducible demonstration, one style of using dinkvm is to put the script and all resources required by the script in one directory tree and then compress that into a single archive file. Using LaTeX as a simple (and half serious) example, the directory tree for this paper with an added script file (DEMO-SCRIPT.sh) looks like this:

```
latexdemo/Makefile
latexdemo/acm_proc_article-sp.cls
latexdemo/build.sh
latexdemo/dot/Makefile
latexdemo/dot/process.dot
latexdemo/jpf-workshop2013.bib
latexdemo/jpf-workshop2013.tex
latexdemo/DEMO-SCRIPT.sh
```

The contents of DEMO-SCRIPT.sh could be the following:

```
#!/bin/bash
[ -f onhost/jpf-workshop2013.tex ] || exit
  # sanity check
cp -r onhost localcopy
cd localcopy
make
cp jpf-workshop2013.pdf /home/knoppix/onhost
```

An archive file with the above contents is small and easy to share. Anybody could then duplicate the demonstration by doing, for example:

```
$ tar xzvf latexdemo.tar.gz
$ cd latexdemo
$ /path/to/dinkvm -dofresh DEMO-SCRIPT.sh
```

It is guaranteed to work, because all resources accessed by the script are in the archive or the Knoppix ISO image, and these can be verified to be unchanged. Everything needed for the demonstration has been verified by the author to be in the resources, including things that might be forgotten like the dot graphics package (for building the figure), and the specific pdf conversion package used. Notice that the shared directory can be used to copy results out to the host before the virtual machine is removed.

3. NET-IOCACHE SCRIPT

For our JPF work, the goal is to produce many demonstrations, each showing that a particular commit passes regression tests. Rather than include everything for each demonstration in a separate archive, we archive only what is common for running the regression tests, because the other information that is particular to each commit naturally can come from our project's Mercurial repositories.

One key point is that we want to check out code from the developer's local copy of the repositories, so that a commit there can be verified *before* pushing it to the shared repositories. This can be achieved by having the developer's JPF project directory be the host directory that is made available to the virtual machine at `/home/knoppix/onhost`. Therefore we have adopted the following directory layout for running net-iocache regression tests using dinkvm.

```
jpf-project-dir/jpf-core/
jpf-project-dir/jpf-core/{files and directories}
jpf-project-dir/net-iocache/
jpf-project-dir/net-iocache/{files and directories}
jpf-project-dir/nioc-inkvm/
jpf-project-dir/nioc-inkvm/one-step.sh
jpf-project-dir/nioc-inkvm/{several *.deb files}
```

And then the developer can copy the demonstration resources and invokes the script like this:

```
$ cd /path/to/jpf-project-dir/
$ git clone https://bitbucket.org/potter/nioc-inkvm.git
$ /path/to/dinkvm -dofresh nioc-inkvm/one-step.sh
```

The nioc-inkvm directory contains all the common resources for running the regression tests, which includes necessary software that is not part of the Knoppix DVD. We were surprised that Mercurial is not on the DVD, but this is problem is easily solved by including the Debian distribution files for mercurial in the nioc-inkvm directory and adding the following line in the script:

```
sudo dpkg -i ~/onhost/nioc-inkvm/*.deb
```

An alternative would be to use apt-get, however putting the specific *.deb file in the archive guarantees that exactly the same resources are used, increasing the chances of repeatable results. It also runs much faster by avoiding network access.

With these fundamental issues resolved, the entire script, shown below, becomes straightforward.

```
#!/bin/bash

# (1) put all this in a unique folder name to make sure
#      path dependencies do not get into the project
datestring="$(date +%y%m%d-%H%M%S)"
mkdir proj-$datestring
cd proj-$datestring

# (2) make a site-properties to match unique folder
mkdir ~/.jpf

cat >~/.jpf/site.properties <<EOF
jpf-core = \${user.home}/proj-$datestring/jpf-core
jpf-net-iocache = \${user.home}/proj-$datestring/net-iocache
extensions+=,\${jpf-net-iocache}
EOF

# (3) Knoppix does not have mercurial and thttpd, so load them
sudo dpkg -i ~/onhost/nioc-inkvm/*.deb

# (4) clone a copy of the mercurial repositories from the host
hg clone ~/onhost/jpf-core
hg clone ~/onhost/net-iocache
```

	host OS	booted VM	VM from snapshot
boot or restore	n/a	25.9	4.0
install *.deb	n/a	19.5	n/a
clone or pull	n/a	107.4	1.3
compile	12.9	11.2	1.4
tests	31.6	39.9	39.4

Table 1: Run times for regression tests (seconds)

```
# (5) compile everything
cd jpf-core
ant
cd ../net-iocache
ant
ant make

# (6) run the tests!!
./bin/regression-tests.sh
```

The entire setup fits conveniently into our JPF workflow. Without changing anything on their machine, a developer can run the above commands. The script copies the latest commit from each repository, ensuring that exactly the files checked into the repository will pass the regression tests. The developer can continue working on the host copy of the files while the test runs in the background.

4. SNAPSHOTS

The time required to run the steps in the above script on a 2.8 GHz 8-core server is summarized in the second column of Table 1. Because the script is easy to invoke and can run unattended in the background, the total of a bit more than three minutes is not too long to wait for believable confirmation that a commit will not break the build. However, faster is always better, especially when debugging iteratively. VM snapshots provide a compromise between exact demonstrations from scratch and fast approximations.

The first step to using snapshots is to invoke `dinkvm` with one extra parameter that specifies a path where `dinkvm` can create a new directory. This directory will be used as a handle to refer to the virtual machine, which will not be removed after the script finishes. After the script finishes, a `-save` command can be issued. For the net-iocache, it could be done like this:

```
$ /path/to/dinkvm -dofresh nioc-inkvm/one-step.sh ./my-vm-handle
$ /path/to/dinkvm -save ./my-vm-handle  ./my-snapshot
$ /path/to/dinkvm -rm ./my-vm-handle
```

After the snapshot has been saved, the third line removes the virtual machine. Now instead of running a script in a freshly booted machine, it is possible to run it in a machine that has been freshly restored from the snapshot. For our net-iocache tests, only pulling the latest changes is necessary, so a simple script like this would suffice:

```
#!/bin/bash
cd proj*/net-iocache
hg pull ~/onhost/net-iocache
ant
./bin/regression-tests.sh
```

And the invocation is the same, except the shorter script is used and the snapshot directory is appended:

```
$ /path/to/dinkvm -dofresh nioc-inkvm/quick-test.sh ./my-snapshot
```

Example run times are shown in the third column of Table 1. Restoring the 778 MB snapshot takes only 4 seconds, which is much faster than the 25 seconds required for booting, and no time is needed to install software. Pulling the latest changes and doing an incremental compile step is also much faster than cloning the entire repository and doing a full compile. Overall, running regression tests inside KVM is reduced to about 46 seconds, which is close to the 31 seconds possible on the host machine.

5. IMPLEMENTATION

The booting step is easy using KVM and a Live CD/DVD like Knoppix with the following short command line:

```
$ kvm -cdrom live-cd-image.iso
```

Knoppix boots without starting the ssh server daemon or other similar services. It makes sense that Knoppix would be configured this way, because on a real machine connected to a network this makes Knoppix more secure. However, it complicates getting a script to run automatically inside the machine after booting.

Our solution uses a special KVM feature to specify the kernel and initial RAM disk (initrd) on the command line. When a Linux system boots, first the kernel and then the contents of a small RAM disk are copied into memory. The first user-mode program that the kernel runs will be on this RAM disk. Therefore it is possible to have complete control over the booting of Knoppix by using a modified copy of the RAM disk. The original contents of the RAM disk are inside the ISO file at `boot/isolinux/minirt.gz`. We added 62 lines to the startup script inside the RAM disk so that it starts a simple server to listen on port 11222 (inside the virtual machine) and take any text sent to that port and execute it as a Bash script.

This solution has worked well but has left us wondering if knowledge of such obscure Linux details is really necessary just to start a simple server. Systems used in cloud computing encounter the same problem about how to take an unchanged template disk image and customize it on first boot. The most popular way to do this seems to be *cloud-init* [10], which runs early in the boot process. It searches the VM's devices and resources on the network for data sources that it then uses as instructions for how to do the customization.[1]

Solutions like cloud-init require that special software be preinstalled in the startup disk. Our solution is a bit unusual in that our startup disk stays unmodified. Also, unlike most solutions, we experimented with avoiding using ssh, because the encryption is unnecessary for our application and the required key management would add some complexity. If ssh is desired, it and its keys can be set up by using the simple script server.

We desired that the script run in an X Windows environment so that it would closely mirror our development environment. This requires more obscure details for the solution. In brief, our script server should be started by the default window manager used by Knoppix, so the configuration file `/home/knoppix/.config/lxsession/LXDE/autostart` needs to be modified during the boot process to actually start our server.

For the shared directory, we use sshfs on Knoppix, which can share files by accessing an sftp-server on the host. The standard way to use sshfs is through an encrypted connection, however we opted for unencrypted use by starting a fresh copy of sftp-server on the host and using the directport feature of sshfs. All of the setup of sshfs can be done using the simple script server, so no modifications to the RAM disk are necessary for this feature.

We use KVM's default user-mode networking, because it does not require any root privileges to set up special devices or change network routing. While it is simple to set up, it does require port forwarding in order for servers inside the virtual machine to be accessed from outside. The dinkvm script automatically finds free ports on the host to forward to the virtual machine so that multiple virtual machines can be started simultaneously. Ports are also chosen and forwarded for ssh access and HTTP access, because it is difficult to set them up later if a sudden need arises during debugging. All the forwarded ports are only bound to the host's localhost interface, therefore they have some security protection because only programs running on the same host have access to them.

The snapshot feature of dinkvm is built on top of the migrate feature of KVM. Essentially it migrates the virtual machine state to a file. Automating this was straightforward, because saving snapshots is controlled by KVM's monitor commands, which can be issued through a telnet interface. Snapshots restore can be specified using KVM command line options.

In summary, the core features added to Knoppix are a script server that starts automatically and a shared directory via sshfs. Even though we were trying to quickly implement a simple solution, just the command line for KVM gives hints that things turned out more complicated than expected.

```
qemu-system-x86_64 -enable-kvm -cdrom \
 /media/sdb2/KNOPPIX_V7.0.4DVD-2012-08-20-EN.iso\
 -net nic,vlan=0,model=virtio -net user,vlan=0,\
 hostfwd=tcp:127.0.0.1:10180-:80,\
 hostfwd=tcp:127.0.0.1:10122-:22,\
 hostfwd=tcp:127.0.0.1:10199-:11222 -net nic,vlan=1,\
 macaddr=52:54:00:12:00:00 -net socket,vlan=1,\
 mcast=230.0.0.1:1234 -m 1024 -vnc :1 -vga vmware\
 -monitor telnet::10197,server,nowait -kernel ./linux\
 -initrd ./minirt.gz -append 'ramdisk_size=100000\
 lang=en apm=power-off nomce libata.force=noncq\
 hpsa.hpsa_allow_any=1 loglevel=1 tz=localtime\
 screen=1024x768'
```

[1]For example, its most widely used data source is to access `http://169.254.169.254/`, expecting that the cloud provider has redirected that IP address to a web server set up specially for the particular booting VM.

6. RELATED WORK

The work closest to dinkvm is Vagrant [6], which shares an almost identical motivation expressed by its creators as "say goodbye to the 'works on my machine' excuse". Vagrant automates creating virtual machines from scratch. It starts with a base box, which must be specially built to include Vagrant's guest tools. After booting, it automatically follows a configuration script (always named Vagrantfile) to run provisioners, which install software and configure the machine. Support for multiple provisioners, such as Chef and Puppet, give flexibility in supporting development environments that must mirror deployed production environments. Vagrant can use various virtual machine monitors (e. g., VirtualBox, KVM, VMware) and can be hosted on MacOS, Windows, and Linux.

Dinkvm can be understood as a simplified Vagrant, where the host is always Linux, the virtual machine monitor is always KVM, the base box is always Knoppix, and the only provisioner is a shell script. This simplicity allows dinkvm to be implemented in about 1,500 lines of shell script. Dinkvm's interface is more tailored to always generating machines from scratch. For example, Vagrant requires an explicit "vagrant destroy" command to first remove previous machine resources. While Vagrant supports suspending virtual machines, it does not have a feature to quickly clone multiple identical environments from the suspended state, like dinkvm can do with its snapshot feature.

The shared directory feature of dinkvm was an attempt to duplicate the *hostfs* feature of User-Mode Linux [3]. It is interesting that Vagrant independently adopted a similar solution, suggesting that it is a natural solution. The snapshot feature in dinkvm was based on the snapshot feature in SBUML [8].

Skytap [2] is a commercial cloud service that can clone test configurations from predefined virtual machine templates for use in testing. By hosting the templates and the virtual machines centrally, developers can remotely access them to have copies of the same development environment without the logistical difficulties of transferring large virtual machine configurations.

Although we do not know of Knoppix being used as a software test environment, it has been used for building specialized software demonstrations, such a distribution for Trusted Computing [9] and another for math applications [5].

7. CONCLUSIONS

Reproducing tests or demonstrations on different hardware can be challenging. Working step by step from scratch using only steps and resources known to be the same is an effective way to produce repeatable results. Reducing the number of steps and using only a few standardized resources helps guarantee that others will be able to independently duplicate the process and achieve the same results.

At its core, the solution presented in this paper uses just two open-source components, KVM and a Knoppix DVD image, both of which can be independently obtained and verified to be unchanged. Knoppix was chosen because it provides many software packages in a single resource, reducing the number of installation steps. Use of these components is simple enough that a user could manually duplicate a demonstration by booting KVM, copying in the resource directory (making sure it is named `/home/knoppix/onhost`), and running the script. We believe it is hard to get simpler than this and are happy that the solution is sufficient for testing net-iocache.

Without changing this core, the `dinkvm` streamlines the interface so that it fits into our workflow with JPF development. Direct from their active work environment, developers can easily invoke tests that run automatically from scratch until completion. Extra features extend this interface to allow interactive control of the virtual machine that is useful for developing and debugging demonstration scripts. We expect this lightweight interface to be useful for other virtual machine applications as well.

Acknowledgments

This work is supported by JSPS *kaken-hi* grants 23240003 and 23300004.

8. REFERENCES

[1] C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weitl, and M. Yamamoto. Software model checking for distributed systems with selector-based, non-blocking communication. In *Proc. 28th Int. Conf. on Automated Software Engineering (ASE 2013)*, Palo Alto, USA, 2013. IEEE Computer Society. To be published.

[2] M. Biddick. The test lab of your dreams. *Information Week*, (1330), 2012.

[3] J. Dike. *User Mode Linux*. Prentice Hall Ptr, 1st edition, April 2006.

[4] R. P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[5] T. Hamada, K. Suzaki, K. Iijima, and A. Shikoda. Knoppix/math: Portable and distributable collection of mathematical software and free documents. In *Mathematical Software—Second International Congress on Mathematical Software (ICMS)*, volume 4151 of *Lecture Notes in Computer Science*, pages 385–390, Castro Urdiales, Spain, 2006. Springer.

[6] J. Palat. Introducing Vagrant. *Linux J.*, 2012(220), Aug. 2012.

[7] R. Potter. `https://bitbucket.org/potter/dinkvm`, 2013.

[8] R. Potter and K. Kato. SBUML: Multiple snapshots of Linux runtime state. *Computer Software*, 26(4):120–137, 2009.

[9] K. Suzaki, K. Iijima, T. Yagi, and N. A. Quynh. Trusted boot and platform trust services on 1CD Linux. *Trusted Infrastructure Technologies Conference, Third Asia-Pacific/Trusted Infrastructure Technologies Conference*, pages 64–71, 2008.

[10] Ubuntu. cloud-init package. `https://launchpad.net/ubuntu/+source/cloud-init`, 2013.

[11] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.