3-2008

A Virtual Computing Laboratory

Joseph P. Kaylor

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Editors: George K. Thiruvathukal, gkt@cs.luc.edu
Konstantin Läufer, laufer@cs.luc.edu

A VIRTUAL COMPUTING LABORATORY

By Joe Kaylor and George K. Thiruvathukal

Many institutions choose to do periodic imaging of computers, which is both painstaking and limiting in terms of keeping software up to date. The authors describe an approach that builds on existing virtualization technologies.

Two of the most difficult administrative tasks in a computer laboratory, classroom, corporate network, or any other area with several computers in use are installing a standard set of software and maintaining updates to it. Computer networks often have more than one hardware platform—in fact, a corporate network might even have one type of machine for its engineers, another type for its accountants, and yet another for customer demonstrations. A university computer laboratory could have Unix machines in the electrical engineering department to support Fortran development and Windows machines in the CS department to support Web development. In any of these cases, it takes time to test software configurations, deploy software and updates, and remedy user misconfigurations (such as malware downloaded from the Internet or other unsupported applications).

Over the years, people have proposed several solutions to these problems, each of which has its own costs and benefits. One approach is *machine imaging*, which lets administrators build well-tested images of operating systems and important software for each type of machine in the network. This approach can tackle, for example, user misconfiguration by establishing a schedule that allows for the rapid introduction of new machines with the same hardware configuration. The great cost of machine imaging is that rolling out updated images requires (sometimes significant) downtime for the machine being imaged—moreover, different hardware platforms require different images. Another problem is that it's somewhat difficult to multipurpose machines—if, for example, the first class in an electrical engineering course uses an early version of Java for classwork (say, version 1.2) and another class in the same room later that day needs a newer version (say, version 1.5), the network administrator must support both versions. To compound the situation, another class in the following semester might need to use Matlab, which means the administrator faces the task of providing a new image one or more times per semester for several classrooms.

Our proposed solution to this problem is the new application of an existing technology: virtualization. With this approach, we can create a machine image for use on heterogeneous hardware platforms that allows for reduced downtime and helps multipurpose the other computers in the network.

Some Basic Benefits

As we mentioned, one of the costs associated with machine imaging is the downtime involved in imaging a machine. With virtualization technology, we can deploy a new machine image to other machines on the network, even while the old machine image is in use: once the new image is deployed to the target machine (or after the current user logs out), we simply boot the new image. The only downtime cost is the shutdown and startup time for the old and new machine images, which is still much shorter than the time it takes to write a new image to a hard disk with conventional machine imaging because the machine can still be used during the imaging process.

Another improvement over conventional machine imaging is the ability to multipurpose machines and better support user bases with differing requirements. With virtualization technology, we can deploy several different images to a single machine. In a university setting, this would mean deploying Unix images for Fortran development, Windows images for Web development, and Linux images for Java development in a single classroom and on a small budget. With conventional machine imaging, we would need at least three classroom setups and a much bigger budget.

Usage Scenarios

Let's look at a few usage scenarios. In Loyola University Chicago's Emerging Technology Laboratory (ETL),

 Copublished by the IEEE CS and the AIP 1521-9615/08/$25.00 ©2008 IEEE

```
<?xml version="1.0"?>
<ServerRepository xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <RepositoryLocation>/media/nfs/vm</RepositoryLocation>
 <ImageEntries>
  <ImageEntry>
   <ImageName>My Image</ImageName>
   <ImageID>7882D784-9268-40a0-A59F-803FA67C6C13</ImageID>
   <Version>0.1</Version>
   <State>
    <Boot>true</Boot>
    <Expires>false</Expires>
    <ExpireTime>2007-11-03T14:26:55.828125-05:00</ExpireTime>
   </State>
   <ImageInformation>
    <MainFileName>ubuntu-server-7.10-i386.vmx</MainFileName>
   </ImageInformation>
   <ServerImageLocation>/media/nfs/vm/ubuntu-server-7.10-i386
   </ServerImageLocation>
   <LocalImageLocation>~/vm2/ubuntu-server-7.10-i386</LocalImageLocation>
  </ImageEntry>
 </ImageEntries>
</ServerRepository>
```

Figure 1. Example configuration. This file contains information about the machine image, including its location on the server, destination on the client, and boot/expiration policies.

experimentation with technologies such as operating systems, clusters, sensor networks, and alternative architectures is very important to the university's mission. With a machine imaging utility that uses virtualization technology, the configuration and deployment aspects of these activities becomes even simpler.

In the case of cluster or grid computing, a challenging task is to deploy a uniform set of system libraries, general machine configurations, and the software that users want to run on the network. With virtualization, the developer can maintain a single machine image and deploy it to several machines in the cluster when it's ready. This, in turn, can lead to increased cluster use because it allows multiple jobs and reduces the need to have more than one cluster accommodate different computing platforms. The developer can also test various software architectures on a smaller set of machines before full deployment.

Another interesting usage possibility is the support of operating system development and experimentation. One challenge in operating system development is how to change underlying code: kernels need compiling, utilities must be deployed, and machines must be rebooted. Dealing with these tasks can slow the overall project and lead to frustration, but if the developer could rapidly and automatically update machine images, the pace would improve. Another challenge in operating system development is supporting and developing multiple machine architectures. With virtualization-based machine imaging, the developer can modify an operating system and then deploy that change to several virtualization architectures automatically. This approach requires less hardware to be purchased and maintained.

New Tools and Techniques

To support the use of virtualization technology in machine imaging, ETL developed a new tool suite. This project not only offered an opportunity to explore a new use for virtualization technology but also a chance to examine modern programming techniques such as test-driven development, test by mock, and the use of design patterns and other platform- and vendor-independent strategies.

For this project, we chose VMware as the virtualization platform, Ubuntu Linux as the host operating system, C#.Net as the development platform, and Windows Vista with Microsoft Visual Studio 2005 as the development environment (http://msdn2.microsoft.com/en-us/netframework/default.aspx). We chose VMware (www.vmware.com) because of its large list of supported guest and host operating system environments, its rich support for external scripting, and its powerful utility toolset. We chose Ubuntu Linux because VMware supports it as a host operating system and because of ETL's support of the open source movement. Finally, we used C#.Net, Windows Vista, and Visual Studio because of developer proficiency with those tools.

One motivation for choosing the .Net platform, other than developer proficiency, was its ability to read and write XML files in a convenient and powerful way. (One of us—Joe—was in favor of this move; George was a bit skeptical at first but agreed to use C# and .Net because he thinks C# is a nice refinement of Java and knows that it can run with the open source Mono project [www.mono-project.com/Main_Page] in the Linux target environment.) The configuration file for the utility in Figure 1 is a crucial

part of the application; other than the interaction with the virtualization software, the manipulation and usage of the configuration file was the most important part of this utility. In .Net, the complex types in the XML are defined with their own class or the `XmlElementAttribute` class; an XML attribute is defined with the `XmlAttributeAttribute` class. In this framework, we could create the code for loading and manipulating the configuration file with three classes in which only the properties containing the configuration values had to be tagged with `XmlElementAttribute` and `XmlAttributeAttribute` attributes. In addition to tagging the properties with these attributes, we tagged each class with the `SerializableAttribute` tag and a total of 10 lines of serialization code. Because of this framework, we didn't have to spend time worrying about writing the code to handle reading and writing XML elements and attributes (see Figure 2).

We used test-driven development, which included writing all code to be testable and capable of participating with dynamic object mocks in unit tests. To support this approach, we used the strategy design pattern heavily throughout the code. This pattern let us decouple the utility design, which, in turn, let us substitute individual pieces of the program with dynamic object mocks and then unit test their interactions with each other.

Figure 3 shows the `ShutdownStrategy` class in the main executable. This class takes as parameters in its constructor an instance of `IVMStatusStrategy` and `IVMBootStrategy`, which poll a virtual machine's current integrity and boot state and provide the ability to boot up and shutdown virtual machines. The purpose of using this strategy pattern was to isolate the decision code for shutting down virtual machines to a small, concise, and testable unit. Figure 4 shows the unit test for this code, with `IVMStatusStrategy` and `IVMBootStrategy` mocked out in a useful library called Rhino Mocks. This test creates a set of virtual machine configurations for shutting down,

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml.Serialization;
using System.IO;
namespace ImageRepository
{
   public class ServerRepository
   {
      private List<ImageEntry> _imageEntries;
      private string _baseRepositoryLocation;
      public ServerRepository()
      {
         _imageEntries = new List<ImageEntry>();
      }
      [XmlElement]
      public string RepositoryLocation {
         get { return _baseRepositoryLocation; }
         set { _baseRepositoryLocation = value; }
      }
      [XmlArray]
      public List<ImageEntry> ImageEntries
      {
         get { return _imageEntries; }
         set { _imageEntries = value; }
      }
      public static ServerRepository Load(string fileName) {
         XmlSerializer serializer = new XmlSerializer(typeof(ServerRepository));
         ServerRepository repository;
         using (FileStream fStream = new FileStream(fileName, FileMode.Open,
           FileAccess.Read)) {
            repository = (ServerRepository)serializer.Deserialize(fStream);
         }
         return repository;
      }
      public static void Save(string fileName, ServerRepository repository) {
         XmlSerializer serializer = new XmlSerializer(typeof(ServerRepository));
         using (FileStream fStream = new FileStream(fileName, FileMode.Open
           OrCreate, FileAccess.Write)) {
            serializer.Serialize(fStream, repository);
         }
      }
   }
}
```

Figure 2. The class responsible for serializing and deserializing a configuration file. It exploits the .Net framework's serialization and XML libraries.

```
using System;
using System.Collections.Generic;
using System.Text;
using ImageRepository;
using ClientActions;

namespace vmclient {
   public interface IShutdownStrategy {
      List<ImageEntry> GetImagesToShutdown
        (ServerRepository myConfig);
      void ShutdownImages(List<ImageEntry> entries);
   }
   public class ShutdownStrategy : IShutdownStrategy {
      private IVMStatusStrategy _statusStrategy;
      private IVMBootStrategy _bootStrategy;

      public ShutdownStrategy(IVMStatusStrategy
        statusStrategy, IVMBootStrategy bootStrategy) {
         _statusStrategy = statusStrategy;
         _bootStrategy = bootStrategy;
      }

      public List<ImageEntry> GetImagesToShutdown
        (ServerRepository myConfig) {
         List<ImageEntry> images = new List<ImageEntry>();
         foreach (ImageEntry entry in myConfig.ImageEntries) {
            if ((_statusStrategy.GetVMState(entry) & VMState.
              VM_BOOTED) > 0) {
               if (entry.State.Expires && entry.State.ExpireTime <
                 DateTime.Now) {
                  images.Add(entry);
               }
            }
         }
         return images;
      }

      public void ShutdownImages(List<ImageEntry> entries) {
         foreach (ImageEntry image in entries) {
            _bootStrategy.ShutdownVM(image);
         }
      }

   }
}
```

Figure 3. `ShutdownStrategy`. This class uses the virtual machine boot and status strategies to determine which images to shut down and provides a method for doing so.

remaining booted, and not booting from an already shutdown state. The mock expectation setup phase sets up the predetermined input and result actions for the `IVMStatusStrategy`. By using dynamic object mocks in this code, we tested not only that this class's output matched the expectations we had based on input but also the interaction of those methods with other classes in the class library. We thus achieved a more restrictive unit test, which helped ensure that this individual class would work as expected, and found we could make requests of other classes.

We also found that some additional advantages came with developing frameworks in a decoupled way other than making the code straightforward to test. When we decoupled classes and sets of classes from each other, we could substitute implementations of those classes quite easily. Our utility currently uses VMware for virtualization, but we can add an additional assembly to the project and use another technology such as Xen Source. If we were to bring in a new virtualization technology, all we would need to do is implement the interfaces for the application's core and write a few unit tests to ensure that the new implementation meets the contracts defined by those interfaces. This architecture also lets us plug applications such as graphical configuration utilities, virtual machine image selectors, and other useful additions into the core utility.

The use of virtual machine images and their deployment via a metadata-driven utility is an exciting new approach to machine imaging and configuration. All the code for the utility described here is available at Google Code (http://vclaboratory.googlecode.com) and can be accessed via Subversion.

Joe Kaylor is a software engineer at a Chicago-area financial consulting company. His technical interests include operating systems, databases, and compilers. Kaylor has a BS in computer science from Purdue University. Contact him at jkaylor@etl.luc.edu.

George K. Thiruvathukal is an associate professor at Loyola University Chicago in the computer science department. His technical interests include parallel/distributed systems, programming language design/implementation, and computer science across the disciplines. Thiruvathukal has a PhD in computer science from the Illinois Institute of Technology. Contact him at gkt@etl.luc.edu.

```
using System;
using System.Collections.Generic;
using System.Text;
using ClientActions;
using ImageRepository;
using NUnit.Framework;
using Rhino.Mocks;

namespace vmclient.Tests {
   [TestFixture]
   public class ShutdownStrategyTests {

      [Test]
      public void TestGetImagesToShutdown() {
         MockRepository mocks = new MockRepository();
         IVMStatusStrategy statusStrategy = mocks.
           CreateMock<IVMStatusStrategy>();
         ImageEntry entry1 = new ImageEntry(
            new Version(1,0),
            new ImageState(true, true, DateTime.Now.
              Subtract(new TimeSpan(1, 0, 0, 0))),
            new ImageInformation(), "path1", "path2",
              "My Image", Guid.NewGuid());
         ImageEntry entry2 = new ImageEntry(
            new Version(1,0),
            new ImageState(true, false, DateTime.Now.
              Subtract(new TimeSpan(1, 0, 0, 0))),
            new ImageInformation(), "path1", path2,
              "My Image", Guid.NewGuid());
         ImageEntry entry3 = new ImageEntry(
            new Version(1,0),
            new ImageState(false, false, DateTime.Now.
              Subtract(new TimeSpan(1, 0, 0, 0))),
            new ImageInformation(), "path1", path2,
              "My Image", Guid.NewGuid());
         ImageEntry entry4 = new ImageEntry(
            new Version(1, 0),
            new ImageState(true, true, DateTime.Now.Add(new
              TimeSpan(1, 0, 0, 0))),
            new ImageInformation(), "path1", path2,
              "My Image", Guid.NewGuid());
         ImageEntry entry5 = new ImageEntry(
            new Version(1,0),
            new ImageState(true, true, DateTime.Now.
              Subtract(new TimeSpan(1, 0, 0, 0))),
            new ImageInformation(), "path1", path2,
              "My Image", Guid.NewGuid());
         ServerRepository config = new ServerRepository();
         config.ImageEntries.Add(entry1);
         config.ImageEntries.Add(entry2);
         config.ImageEntries.Add(entry3);
         config.ImageEntries.Add(entry4);
         config.ImageEntries.Add(entry5);
         Expect.Call(statusStrategy.GetVMState(entry1)).
           Return(VMState.VM_BOOTED).Repeat.Once();
         Expect.Call(statusStrategy.GetVMState(entry2)).
           Return(VMState.VM_BOOTED).Repeat.Once();
         Expect.Call(statusStrategy.GetVMState(entry3)).
           Return(VMState.VM_BOOTED).Repeat.Once();
         Expect.Call(statusStrategy.GetVMState(entry4)).
           Return(VMState.VM_BOOTED).Repeat.Once();
         Expect.Call(statusStrategy.GetVMState(entry5)).
           Return(VMState.VM_NONE).Repeat.Once();
         mocks.ReplayAll();
         List<ImageEntry> results = new ShutdownStrategy
           (statusStrategy, null).GetImagesToShutdown(config);
         mocks.VerifyAll();
         Assert.AreEqual(1, results.Count);
         Assert.Contains(entry1, results);
      }

      [Test]
      public void TestShutdownImages() {
         MockRepository mocks = new MockRepository();
         IVMBootStrategy bootStrategy = mocks.
           CreateMock<IVMBootStrategy>();
         Guid imageID1 = Guid.NewGuid();
         ImageEntry entry1 = new ImageEntry(new Version(1,
           0), new ImageState(), new ImageInformation(),
            "path1", path2, "My Image", imageID1);
         ImageEntry entry2 = new ImageEntry(new Version(1,
           1), new ImageState(), new ImageInformation(),
            "path1", path2, "My Image", imageID1);
         List<ImageEntry> images = new List<ImageEntry>();
         images.Add(entry1);
         images.Add(entry2);
         bootStrategy.ShutdownVM(entry1);
         LastCall.Repeat.Once();
         bootStrategy.ShutdownVM(entry2);
         LastCall.Repeat.Once();

         mocks.ReplayAll();
         new ShutdownStrategy(null, bootStrategy).
           ShutdownImages(images);
         mocks.VerifyAll();
      }
   }
}
```

Figure 4. Test fixture for `ShutdownStrategy`. The unit test framework used is NUnit, and the dynamic mocks framework is Rhino.Mocks.