

Applications and Enhancements of Featherweight Virtual Machine (FVM)

A THESIS PRESENTED

BY

HARIHARAN KOLAM

TO

THE GRADUATE SCHOOL

IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

STONY BROOK UNIVERSITY

MAY 2008

Stony Brook University

The Graduate School

Hariharan Kolam

We, the thesis committee for the above candidate for the
Master of Science degree,
Here by recommend acceptance of this thesis.

Professor Tzi-cker Chiueh, Thesis Advisor
Computer Science Department

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

Applications and Enhancements of Feather Weight Virtual Machine (FVM)

by

Hariharan Kolam

Master of Science

in

Computer Science

Stony Brook University

2008

Featherweight virtual machine [11] (FVM) is an OS level virtualization technique on Microsoft Windows OS. Under FVM architecture, each virtual machine is created using the same state as the host machine. The virtual machines are logically isolated using namespace virtualization, resource copy on write and IPC confinement.

The key idea of FVM is access redirection and copy on write, which enables each VM to read the base environment from the host machine but write into the FVM private environment. FVM identifies various communication interfaces and confines its scope on a per VM basis.

We describe the design and implementation details that we followed to virtualize Windows installer, which is an operating system resident installer service. Windows installer service performs installation duties (files/registry updates) on behalf of applications. Windows clipboard forms a communication medium (a scratch pad) available to Windows and all running applications. We also describe the design/implementation details for visualizing Windows clipboard.

We also present an application of FVM framework: binary server. Binary server allows Windows desktop to share binaries that are centrally stored, managed and patched. The shared binaries are launched in a VM whose runtime environment is imported from a central binary server. We describe the procedure to customize the generic FVM framework to accommodate the needs of these applications and present experimental results to demonstrate their performance and effectiveness.

To
My Family

Table of Contents

List of Tables	vi
List of Figures	vii
Publications	ix
1 Introduction	1
2 Related Work	7
3 FVM Enhancements	9
<i>3.1 Overview</i>	9
<i>3.2 Windows Installer Virtualization</i>	10
<i>3.3 Clipboard Virtualization</i>	13
4 Shared Binary Server	15
<i>4.1 Introduction</i>	15
<i>4.2 Overview of FVM based shared binary server</i>	16
<i>4.3 Design and Implementation</i>	16
<i>4.4 Performance Evaluation</i>	20
5 Conclusion and future work	25
Bibliography	26

List of Tables

1: SHARED BINARY SERVER REDIRECTION LOGIC	19
2: NUMBER OF REGISTRY/FILES REDIRECTED	23

List of Figures

1: OS LEVEL VIRTUALIZATION VS HARDWARE LEVEL VIRTUALIZATION	2
2: FVM VIRTUALIZATION LAYER AT SYSTEM CALL INTERFACE LAYER	3
3: FVM VIRTUALIZATION TECHNIQUE.....	10
4: WINDOWS INSTALLER VIRTUALIZATION TECHNIQUE.....	12
5: SUMMARY OF APPLICATION DEPLOYMENT ARCHITECTURES	15
6: BINARY SERVER ARCHITECTURE.....	17
7: COM REDIRECTION TECHNIQUE IN SHARED BINARY SERVER	18
8: BINARY SERVER EVALUATION-1	21
9: BINARY SERVER EVALUATION-2.....	21
10: BINARY SERVER EVALUATION-3.....	22

Acknowledgements

First of all, I sincerely wish to thank Dr. Tzi-cker Chiueh for his guidance and support all through the project. I would also like to thank all the present and past colleagues at Rether Networks Inc. In particular, Dr. Lap-chung Lam for his insightful advises and helps on my thesis work and Sheng-I Doong, President of Rether Networks, for her kind support. I also want to give my thanks to Dr. Yang Yu for mentoring me and helping me bootstrap on my thesis work. I would also want to give my thanks to Subhadeep Sinha and Zhiyong for their help.

Publications

- Yang Yu, Hariharan Kolam Govindarajan, Lap-Chung Lam and Tzi-cker Chiueh, “Applications of a Feather-weight Virtual Machine”, to appear in Proceedings of the 2008 International Conference on Virtual Execution Environments (VEE’08), March 2008

Chapter 1

Introduction

1.1 Virtual Machine – an overview

Virtual machine (VM) is a technology that enables multiple execution environments on a single physical machine. VM does typically hide the physical characteristics of a computing resource from its users. Thus a single physical resource (for example, a server) could be made to appear as multiple virtual resources. VM's have been extensively used for the following purposes,

- *Server consolidation*: Consolidate the workloads of several under-utilized applications to fewer machines (perhaps a single machine too). Benefits include saving on administration costs and management costs.
- VM's could be used to provide secure, isolated *sandboxes* for running untrusted applications. Virtualization *is* an important concept in building secure computing platforms. Since, the VM's could be used to isolate what they run, they provide *fault and error containment*.
- VM's makes software easier to migrate, thus aiding in system (and application) mobility.
- VM's can be used to create varied test scenarios, and can lead to some very imaginative, effective quality assurance.

To support virtual environments with software approaches, a virtualization layer must be placed at certain levels along the machine stack. This virtualization layer thus partitions physical machine resources and maps the virtual requests from a VM to physical requests. This virtualization layer could be hardware embedded, in which case the hardware provides architectural support that facilitates building a *Virtual Machine Monitor* and allows multiple Operating Systems (called the guest OS's) to run in isolation. The virtualization layer could also emulate the entire instruction set of a VM in software which is called the *Instruction Set Architecture (ISA)*. *Hardware Abstraction Level (HAL)* virtualization exploits the similarity between the architectures of the guest and host machine, and directly executes certain instructions on the native CPU without emulation. OS-level virtualization partitions the physical machine's resources at the operating system level. This means all the OS-level VM's share a single operating system kernel. Fig 1.1 depicts the difference between OS level virtualization and Hardware level virtualization. *Hypervisor* (or the *Virtual Machine Monitor*) is the name given to the virtualization platform that allows multiple operating systems to run on host computer at the same time. Hypervisor's could be implemented to run directly on a bare hardware platform (typically as a control program). VMware ESX server, Citrix XEN server are typical examples for such configurations. Hyper visor could also software that runs within an operating system environment. The guest operating system in this case runs on a third level above the hardware (e.g. VMware Workstation, VM Fusion etc.)

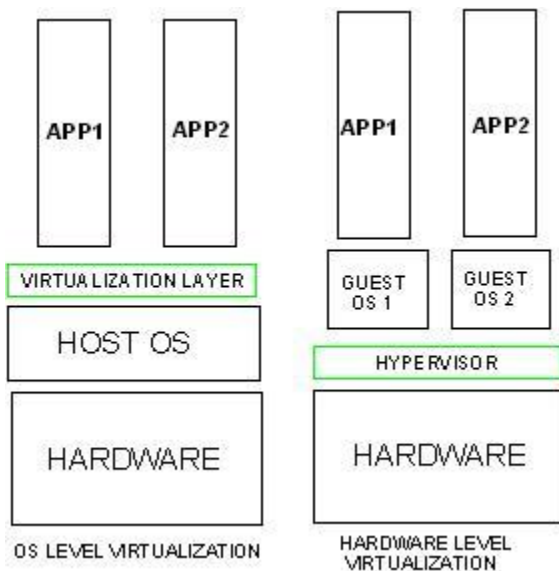


Figure 1.1 OS level virtualization vs. Hardware level virtualization

1.2 Feather-Weight Virtual Machine (FVM)

FVM is a Windows based OS level virtualization technique, which is specifically designed to reduce the invocation latency of a new VM and to scale to a large number of VMs by minimizing per-VM resource requirement. The virtualization layer, in FVM, virtualizes the namespace by renaming the system resources, which forms the key idea of FVM. This renaming takes place at the OS system call interface. Microsoft Windows supports numerous namespaces for various system resources (system resources include files, registries, kernel objects, network address, daemon services etc). Whenever a process makes a system call to access any of the above-specified resources, the FVM layer manipulates the names of the resource. This renaming ensures that the namespaces visible to processes in one VM is disjoint to those visible to a process executing in a different VM. This further ensures that, two VMs never share any resources and therefore cannot interact with each other directly. For example, when a process in a VM (say VM1) tries to access a file named *“/foo/bar”*, the virtualization layer redirects access to a different file (say *“/VM1/foo/bar”* on the host workspace). This virtual to physical mapping is transparently performed inside the virtualization layer which, in case of FVM, is at the system call interface/system call library interface layer.

All the VMs share the host OS’s kernel-mode component, including the hardware abstraction layer, device drivers, OS kernel as well as system boot components. The file system image is also shared by default. Each new VM starts with exactly the same operating environment as the current host. Therefore, both the startup delay and the initial resource requirement for a VM are minimized. Since, the resource virtualization is performed by simply renaming system call arguments instead of complicated resource mappings or instruction interpretations, an application’s runtime performance in a VM is also improved. Duplicating the resources to each VM’s partition involves significant resource costs on the physical machine (along with the VM initialization and termination overhead). FVM typically shares most resources with the host environment. Private

copies are created only when a resource is to be modified by the VM. Figure 1.2 pictorially depicts the FVM virtualization architecture.

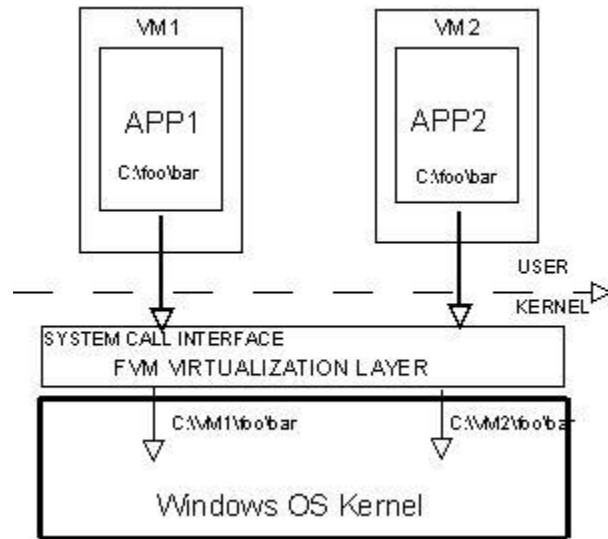


Figure 1.2: FVM virtualization layer at OS system call interface.

FVM virtualization layer is implemented by intercepting Windows system calls, which are exposed to the user-mode applications through a set of user mode dynamic linked libraries (DLL's). The intercepting mechanism is implemented mostly in the kernel (it is difficult to be subverted or bypassed than the user mode interceptions). However, some of the interceptions are implemented in the user space too. The reason for user mode interception being, some of the system calls like those managing daemon service, GUI window and network interface either do not have kernel mode interface or have a kernel mode interface with no clear documentation. The files, registries, kernel objects are virtualized inside the kernel. The kernel mode interceptions are implemented using a driver, which modifies the system call entry point in the *System Service Dispatch Table* (SSDT) within the kernel. The user mode component is a DLL that modifies the library function entry point using the Detours library [1].

1.2.1 User mode interception

Typically, Windows OS provides a set of standard API's based on which user processes request OS services. The API's are generally implemented as dll's (like *kernel32.dll*, *user32.dll* etc). These dll's are the most common user level interception points.

We rely on *Detours* library for user level interceptions in FVM. *Detours* library replaces the first few instructions of the target API function with an unconditional jump to the user provided function, called a *detour function*. The replaced instructions from the target function are saved in a *trampoline function*, with an unconditional jump to the rest code of the target function. Thus, a process's call to a target API is routed to the detour function, which can call its *trampoline function* when it requests service from the original target API function. We inject our custom DLL to the target process's address space.

When this DLL is being loaded, its initialization routine is invoked to perform *Import Address Table (IAT)* modification or function rewriting. The custom DLL, which we inject, provides the intercepting function that has the same function signature as the target function and is called when the target function is called.

1.2.2 Kernel mode interception

A process makes system calls to request OS services, and the system calls may deliver the user requests to certain device drivers, which process the requests and return the result to the caller. Therefore, we can intercept system calls or driver functions to change the system behavior. Compared with user-land interceptions, interception at kernel level is more reliable because user application processes cannot bypass it.

To make a system call on Windows NT kernel, a process loads the EAX register with the ID of the system call and executes INT 2E or SysEnter instruction. This causes the calling thread to transit to kernel mode and to execute the system call dispatch routine. The dispatch routine locates the address of the system call stored in the Windows system call table, called System Service Dispatch Table (SSDT), and then starts execution of the system call. The SSDT stores the addresses of all the Windows system calls, indexed by the system call ID. To intercept a system call, in the FVM implementation, we replace the system call's SSDT entry with the address of our own function, which can call the original system call function plus pre and post-processing. The issues and alternatives of the currently used kernel mode interception in FVM are evaluated in details in Yang Yu's Ph.D. dissertation [12].

1.3 Contributions

This thesis covers a couple of enhancements to the core FVM – the Windows installer service virtualization Windows clipboard virtualization. The later part of the thesis also describes, in details, an application of FVM – Shared Binary Service (architecture, implementation and evaluation).

1.3.1 FVM enhancements

The isolation between multiple VM's and between a VM and the host environment makes virtual machines an effective platform to support fault tolerant and intrusion tolerant applications. Any changes made to the host environment (file/registry) by a process running inside the VM (say VM1) should be “charged” to VM1. This would ensure that when VM1 is deleted, all the changes made by it are cleaned up (without any side effects). In addition, the isolation mechanism needs to ensure that all inter-process communication interfaces are virtualized and confined to processes on a VM basis. The Windows installer service virtualization essentially ensures that all changes made by the Windows installer service process, on behalf of an application installation initiated from a VM, are “charged” to that VM.

Windows installer service is an operating system component that describes a standardized format for applications (known as Windows installer format). This service performs the installation duties on behalf of the applications. An *installable resource* (or

a *resource*) is defined as a file, registry key, shortcut, or any other piece that an installer typically delivers to a computer.

Though FVM has virtualized daemon (service) process management to start certain daemon processes in each VM, some daemon processes still need to be shared between all VM's. These daemons cannot be started on a per-VM basis because they are either critical components of system booting or they have some close dependencies with some kernel drivers. Thus, VM's *IPC* to these daemons should be enabled so that the processes inside the VM execute properly. Windows installer is one such daemon process, which needs to be shared across multiple VM's. All of the files/registry (resource) modification during a *Microsoft Software Installer (MSI)* based application installation is performed by the installer service. To charge the installable resource to a VM, that initiated the installation, we serialize (associate) the access to the installer service on a per-VM basis. We describe the design, implementation and analysis of the approach we followed as part of this thesis.

The Windows Clipboard is one additional inter-process communication method. Clipboard is like a scratch pad available to Windows and all running applications. It allows pieces of information to be temporarily stored and later retrieved by other applications. Thus, Windows Clipboard forms a means of information sharing between processes. To ensure that multiple VM's are isolated from each other, we need to ensure that the Clipboard is virtualized. This essentially means that a process from one VM (say VM1) should not see the contents that a process from a different VM (say VM2) has stored in the clipboard.

Windows Clipboard has the following features,

- Only one item stored at a time
- Each new copy replaces the last
- Clips are not remembered between sessions

A memory object on the clipboard can be in any data format, called a clipboard format. Each format is identified by an unsigned integer value (defined in *Winuser.h*). A window can place more than one object on the clipboard, each representing the same information in a different clipboard format. The clipboard formats are generally oblivious to the users.

The clipboard virtualization creates an independent clipboard view on a per-VM basis.

1.3.2 Shared binary service (A FVM application)

Shared binary service requires the end user machine to fetch all its application binaries from a central server. We need to provide applications installed on a central shared binary server but executed on a client machine. Clearly, all the application files, registry entries, configuration files, COM objects, DLL's etc which are required for execution of the application physically reside on the server.

FVM's OS virtualization technique could be leveraged to implement the shared binary service application. With shared binary service, it is possible to reap benefits of

both client server computing as well as centralized resource management (one of the key features of a *thin-client* computing).

As part of this thesis, we also show how a generic FVM framework could be extended to accommodate the needs of shared binary service application. We have tailored the FVM namespace virtualization technique to implement the Shared binary server application. We describe the architectural details, implementation details and experimental results that demonstrate its performance and effectiveness. We also evaluate some of the optimizations to FVM based shared binary server architecture like client side file/registry caching.

Chapter 2

Related Work

Application deployment architecture

Deployment of applications on an end user machines could be done in one of the following methods: *local installation, network boot, thin client installation, OS streaming, application streaming and shared binary service.*

In the local installation architecture, application binaries are installed and executed on the end user machines. In this architecture, using network-wide application installation/patching tool, we could reduce the application management cost. Network boot architecture allows a diskless end user machine and enables it to boot a kernel image located on a remote server and thus run OS as well as application code on the machine's local CPU. *Citrix Ardence* [2] refers this technology as disk streaming. The technology that *Ardence* uses enables *Citrix* servers to boot from a centralized disk image files stored on a file server instead of each server having its own drive. An end user machine boots from the network through *Preboot Execution Environment* (PXE) and then accesses the virtual disk via a protocol called BXP. However, because of hardware dependencies, the virtual disk of each end user machine may be different from one another and thus require extensive customization. This architecture offers reduced application management costs as the binary installation and management are centralized.

In thin client computing architecture, such as Microsoft Terminal Service [3], application binaries are installed, maintained and executed on the server's CPU. An end user machine displays the result of application executions and replays user inputs via a special protocol such as *Remote Desktop Protocol* (RDP). The application maintenance costs are considerably lowered due to centralized binary installation and execution. This architecture, however, incurs large application runtime latency and higher network traffic load. Since, each of the application instances are executed on the server, the server should be a high-end machine with adequate resources to cater to all the clients. This architecture has a disadvantage that it does not leverage the compute power of the local end user machine and instead treats the same as dumb terminals. In comparison to the network boot architecture, thin client architecture is more generic (no hardware dependencies) as the application executes on the remote server CPU and only the display is exported to the end user machine.

In an OS streaming architecture [4, 5], is similar to the network boot architecture where the OS as well as specific sets of applications are configured into a hardware level virtual machine image stored on a central server. The end user (client) machine runs one of the VM images on a *Virtual Machine Monitors* (VMM). The VM image has less

hardware dependencies as it is isolated by the VMM from the underlying physical hardware. However, in practice, the hardware dependencies are transferred to the VMM, which is not necessarily the best party to deal with hardware dependencies because of its emphasis on minimal code base. Streaming both OS as well as applications embedded with it does reduce the operational costs and provides centralized management flexibility. However, OS streaming does involve significant network overhead (in most cases reduced by using stripped down versions of network protocol stack [2]).

In the *application streaming* architecture [6, 7, 8, 9], application binaries and configurations are bundled into self-contained packages, which are stored and maintained centrally. An end user machine running a compatible OS fetches these packages from the server and runs them directly without local installations. Each such package runs in a virtual environment, and is cached on the local machine whenever possible. *Thininstall's* technology allows Windows application to be processed into an executable file that can see its own version of Windows registry, file system and DLL's. This essentially ensures that incompatible applications can run on the same machine without conflict. This architecture offers the advantages of both centralized application management and localized application execution on end user machines. However, most of the applications (like Microsoft Office suite) support on demand installation of features (called as *advertisement* in Microsoft lingo). Essentially the installation of a feature is triggered only when the user or application uses/activates it. Packaging of such applications would typically warrant tracing through all possible execution paths of the application to enable package all features. This task is very specific to each supported application. Softricity [14] solves this problem by creating a primitive application cache file (by monitoring the default application installation on the server). The advertised feature is downloaded from the Softricity server (pre-configured) and it becomes part of the user's application cache file when ever the application or user uses/activates it. However, an issue with packaging is that upon subsequent software updates at the server, the package has to be refreshed in all the clients (otherwise, the application would continue to use older versions of files/registries).

The *shared binary service* [10] architecture, which is widely used in the UNIX world, is similar to application streaming except that it does not require explicit application packaging, and it allows resource sharing among packages. More concretely, applications are installed on a shared binary server, and then exported to all end user machines through a standard network file-sharing interface. When an application is executed, accesses to binaries, configuration files and registry settings are redirected to the central binary server. *Shared binary service* architecture ensures that the client is always updated with the latest updated binaries/DLLs from the server (the access is always redirected to the central binary server).

The interception methodology used for shared binary server application is similar to the FVM interception technique [12]. The FVM interception technique is tailored to redirect resource access to the server, over the network. Shared binaries are set up as a SMB share on the server.

Chapter 3

FVM Enhancements

3.1 System overview (Feather-weight Virtual Machine)

Virtual Machine (VM) is typically defined as sets of user-mode processes accessing underlying hardware through multiple interfaces at various levels like library, system calls, I/O etc. Support for multiple execution environment (VMs) on a single physical machine would warrant modifying these interfaces so that access requests from different VMs can be mapped to requests to different OS objects (to ensure that there is no interference between VMs). This modified interface is called the virtualization layer. This layer determines the scalability, runtime performance and isolation. In general, the isolation between the VMs is complete when the virtualization layer is close to the hardware. However, when the virtualization layer is very close to the hardware, scalability and performance of the system is deteriorated as many system resources are duplicated without sharing.

The *Feather-weight Virtual Machine* (FVM) project is designed to facilitate sharing (OS kernel, file system, hardware) in order to improve the scalability and runtime performance, while still maintaining a certain degree of isolation. The virtualization layer in FVM is at the OS's system call interface, as shown in Figure 3.1. All the VMs share the host OS's kernel-mode components, such as OS kernel and device driver. Moreover, critical system daemons and the file system image are also shared by default. Each new VM starts with exactly the same operating environment as the current host OS. Therefore, both the startup delay and the initial resource requirement for a VM are minimized. Renaming system call arguments instead of device emulation or instruction interpretation performs the virtualization. Therefore an application's runtime performance in a VM may also be improved.

The FVM virtualization layer observes all the access requests from user-mode processes. As a result, it can redirect access to the same OS object from different VMs to requests against different versions of the same OS object. FVM uses namespace virtualization and resource copy-on-write to implement the access redirection and isolation between different VMs. When a new VM (say vm1) is created, it shares all the system resources (disk files, system configurations, etc) with the host machine. Later on, when different types of requests from a process p in the VM pass through the FVM layer, these requests can be redirected as follows:

- If p attempts to create a new file /a/b, the FVM layer redirects the request to create a new file vm1/a/b.
- If p attempts to open an existing file /a/b, the FVM layer redirects the request to open a file vm1/a/b. If file vm1/a/b exists, no further processing is made in the FVM layer; otherwise, the FVM layer checks the access flag of the open

request. If the access is “open for read”, the request will go to the original file /a/b; if it is “open for write”, the FVM layer copies /a/b to vm1/a/b, and then redirects the request to open vm1/a/b again.

- If p attempts to read or write an existing file, the FVM layer passes the request through without additional processing, because read/write request is based on a file handle, which is returned by a previous open request. If the open request has been redirected, all the subsequent read/write requests on the same file handle are redirected inherently.
- If p attempts to delete an existing file /a/b, the FVM layer only marks the file as deleted by adding its name /a/b to a per-VM log. The file /a/b is not deleted from the file system.
- If p attempts to make any types of inter-process communications, such as sending window message, to another local process, the FVM layer examines the two processes and blocks the communications unless they are running in the same VM.
- If p attempts to access and communicate with certain devices, the FVM layer denied the access unless it is permitted by a policy.

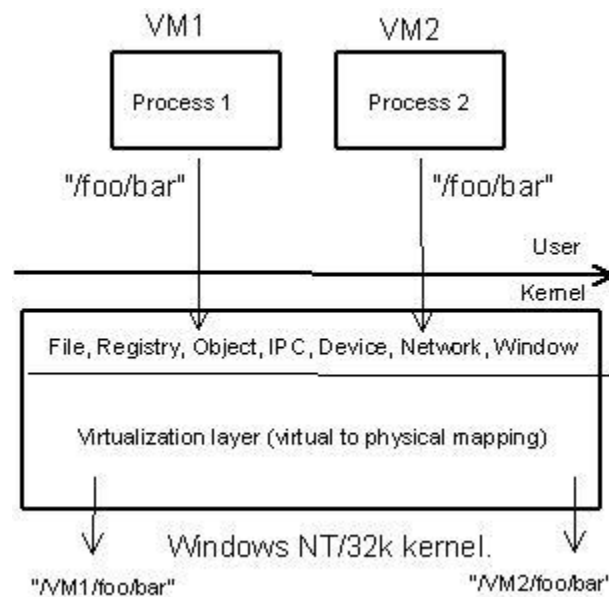


Figure 3.1: FVM namespace virtualization technique.

3.2 Windows installer service virtualization

Installation (or setup) of an application is the act of putting the program in a computer system so that it can be executed. An application is usually condensed into a module that could be distributed. In order to be used, the module must be unpacked and relevant information must be placed at correct places on the computer. An installer is a specialized program that automates most of the work required for installation. *Windows Installer* (MSI) is an engine for installation, maintenance and removal of software on Windows OS. The installation information, and often the files themselves, are packaged in installation packages, loosely based on relational databases, commonly known as MSI packages. There are other different tools available on Windows OS for creating installer

programs. This includes Install Shield [15], Install Anywhere [16], Wise [17] and Script Logic [18]. Most of these tools create MSI packages as well as their own proprietary executables. The installation/un-installation and maintenance of MSI packages are performed by the Windows Installer service program. The changes to the computer due to unpacking the MSI installer package is performed by this Windows Installer service daemon (create/update of files/registry entries etc). Proprietary installers, generally, do not rely on Windows Installer service program to perform the installation changes. The changes are rather specific to the tools that create the installer program. The tools also create uninstall scripts which would be potentially be called during un-installation to undo the changes created by the installation.

The effect of installation of an application inside a VM should be confined to that VM i.e., all resource changes made by the installer should be accounted to the VM inside which the application that initiated the installation is running. However, Windows installer service process is one of the critical components of system booting and hence cannot be started on a per-VM basis. Since the installer service performs the installation duties on behalf of the application, the resource changes made by the installer service have to be charged to the VM. We use the following method for to ensure that the installation carried by the Windows installer service is virtualized. (Figure 3.2 is a pictorial depiction of the same),

- During the FVM driver loading, we determine the PID of the installer service process. To get the PID of the installer service, we enumerate all the system processes and find a process whose parent is the process Services.exe (the service control manager) and whose image name is Msiexec.exe.
- Whenever a process from within an FVM (VM1) tries to communicate to the installer service, we associate the installer process into VM1. VM1 now owns the installer service process. This would ensure that any changes made by the installer service process are confined to VM1. Upon completion of the installation activity, the installer service process is removed from VM1.
- Any interim requests to access the installer service process as long as it is associated with a VM are rejected (unless of course the request comes from a process running inside the VM owning the installer service). Thus, the access to the installer service process is essentially serialized.

The installable application communicates with the windows installer service using a utility called msiexec.exe. Msiexec.exe utility is used for install/uninstall activity. We intercept this communication between the application and the installer service to associate the installer service process with a VM and ensure serialization of access.

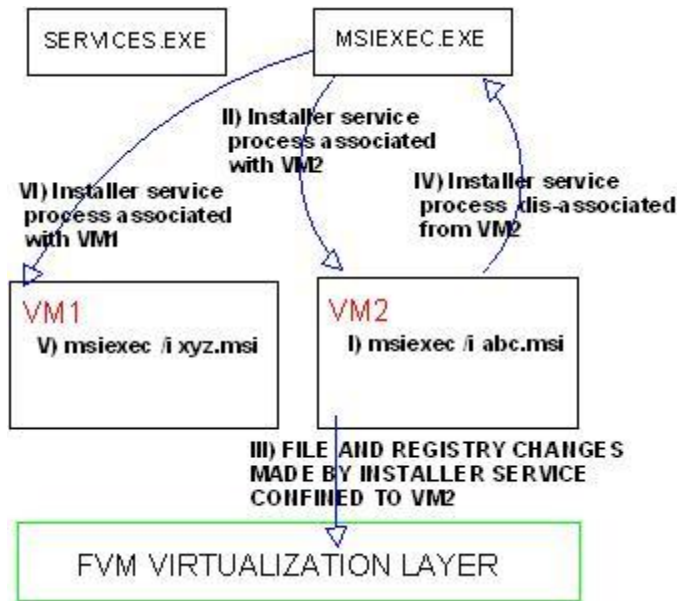


Figure 3.2: Windows Installer virtualization technique.

The above procedure would ensure that any changes made by the installer service directly (as part of installation) are localized to a VM (where the application initiating the installation resides). However, the above technique is not complete when the installer registers (or uses already registered) *Common Object Model (COM)* classes. COM (*Component Object Model*) is a platform independent, distributed and object-oriented system for creating binary software components that can interact. COM server is a *Remote Procedure Call (rpcss)* service. And all resource manipulations (registry/file changes/creation) that the COM component does on behalf of the COM client are performed by this service. So, virtualizing *rpcss* ensures that the Windows Installer virtualization is complete. The *rpcss* virtualization also helps to complete the Windows Clipboard virtualization procedure. This will be described in **section 3.3**

We verified our Windows Installer virtualization method described above by installing applications (installers) created by differed tools (which includes, Install Shield created installers like Source Insight 3.1, custom installers like Adobe acrobat reader 8.1, Winzip and thunderbird. We also tested out a bunch of MSI installer programs like Apache 2.2, Log parser, Folder view etc. We also tested out installing the whole Microsoft Office suite inside the FVM. Installations were initiated from within the FVM and it was verified that the changes due to the installation were localized to the VM initiating the installation. Some of the installers like Microsoft Office, Acrobat Reader suite registers and uses COM object as part of their installation process. The filesystem/registry changes made by the COM server (*rpcss* – not virtualized) could not be localized to a VM (some sideeffects/leftovers after deleting the VM inside which the application installation was triggered). *Rpcss* virtualization does ensure that the side effects are eliminated.

We tested and verified different installers created from different tools on Windows 2000 OS. The virtualization approach works without any side-effects to the host machine. However, Windows XP has introduced the concept of application binary *prefetching* which modifies the way Installer service works. The Windows Installer service virtualization needs to be extended to accommodate the same.

3.3 Clipboard virtualization

Window clipboard is a convenient tool to copy or move information between different processes. To use the clipboard, the user generally needs to just select the information (text or image) that he wants to copy or move to a different place. The information is first copied onto the clipboard (when the copy is initiated at the source location) and at the destination, the information is copied out (paste) of clipboard to the destination location.

There are three ways that you can copy and cut to the clipboard, and there are three ways that you can paste from the clipboard.

- Using the File->Edit menu
- Key board shortcuts (Cntrl+C for copy, Cntrl+X for move and Cntrl+P for paste)
- Context menu. (In general, right mouse click displays the context menu options)

Windows Clipboard is thus, a very simple way to share data between processes. Virtualizing Windows Clipboard would ensure that the information sharing is confined to processes running inside a particular VM. We intercept the 2 API's (user level interceptions using detours), GetClipboardData() and SetClipboardData() defined in user32.dll to implement Clipboard virtualization. The following is the procedure that we use to virtualize,

- FVM service daemon is created whenever the FVM driver is loaded. We create a shared memory mapping as part of this service daemon (using CreateFileMapping() defined in kernel32.dll) which we use to share store clipboard data on a per VM basis.
- We intercept SetClipboardData() API and store the clipboard data along with the format information on a per VM basis in the shared memory created. We also keep track of the Clipboard data format information in addition to the
Upon clipboard data query from within a VM (say VM1), we return the contents of the clipboard pertaining to VM1.

Apart from the Clipboard that Windows provides, a lot of custom applications implement their own clipboard variants (generally as out-of-proc COM components). Out-of-proc COM components are generally implemented as EXE's and is shared between multiple applications. Office suites like Microsoft Office, Open office support clipboard which could contain multiple items (Standard Windows supported can contain only one item inside the clipboard at a time). They usually implement their clipboard using out-of-proc component (as a clipboard server) to manage multiple data items. Thus, COM server virtualization is essential to complete clipboard virtualization.

The COM server (rpcss) virtualization was recently implemented successfully by Zhiyong as part of other FVM enhancements. In summary, the implementation involves creating multiple rpcss services (one per VM instance). This service is created during the

FVM creation and destroyed when FVM is destroyed. Also, the boot time start dependency of rpcss service is removed and the service is started inside each VM (upon a VM start). COM server virtualization essentially solves many associated problems like the installer virtualization issue mentioned above, clipboard virtualization issue as well as the shared binary server issue (using out-of-proc COM objects) which is described in Chapter 4.

Chapter 4

Shared Binary Server- An Application of FVM

4.1 Introduction

OS-level virtual machines powered by the FVM framework make an effective computing platform for many useful applications on Windows server and desktop environment, such as malware sand boxing, intrusion tolerance and analysis, access control, and others that require an isolated but realistic execution environment. Here we present the design, implementation and evaluation of one such application – Shared binary server.

A shared binary service is an application deployment architecture under which application binaries are centrally stored and managed, and are exported from a server to end user machines. With centralized management, this application deployment architecture greatly simplifies software patching/upgrade/repair and license control in a corporate environment without suffering performance penalty and scalability problem of the thin client-computing model. We try to describe how to customize the generic FVM framework to accommodate the needs of the shared binary server architecture and present experimental results to demonstrate their performance and effectiveness.

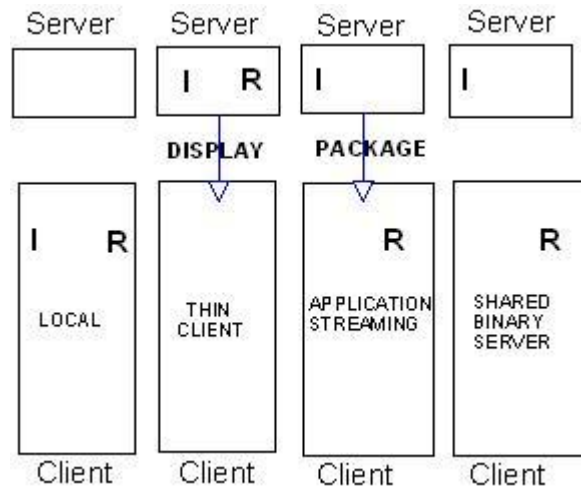


Figure 4.1: Application deployment architectures.
(I: Install, R: Execution)

4.2 Overview of FVM based Shared binary server

The shared binary server is widely used in the UNIX world. End user machines typically mount binary files exported by a central binary server on local directories and execute them locally. However, in Windows world, this scenario is a bit tricky. Most applications are not designed to be loaded from a shared repository and run locally. Instead, they are designed to run on the machine on which they are installed.

Consequently, whenever an application runs, it tries to locate its operating environment like registries, libraries, configuration files etc from the local machine.

To enable a Windows client machine to execute a program physically stored on a binary server, the program's operating environment has to be redirected from the local machine to the binary server. However, when accessing a local file containing the input/output data, the access should not be redirected.

FVM architecture enables us to redirect access of a process running in a VM to the VM's private workspace. We extend this to framework to support redirections on the binary server client. We modify the redirection logic on the binary server client as described above. A process started from the executable on the binary server is associated with a special VM (with redirection logic modified) on the binary server client. To distinguish between accesses to local/input output data and the application configuration data, we rely on the typical application installation patterns (based on heuristics). An application installation, usually, updates only a few directories like the program directory (*C:\Program Files*), the system directory (*C:\winnt\system32*) and the configuration directory (*C:\Documents and settings\All users*). The installation, usually, modifies the registry keys located at *HKLM\Software*. Users rarely store their personal data at these locations. The binary server is implemented over these heuristics and testing with a variety of Windows applications suggests that this heuristic is pretty reliable and reasonable.

4.3 Design and implementation

Figure 4.2 pictorially depicts the system architecture for the proposed shared binary server for Windows based end user machines. Under binary server architecture, users start the application process on their local machine by executing the application's executable file stored on the binary server. The binary server is accessed using remote file share/access protocol like CIFS/SMB. The process creation on the binary server client is intercepted. We create a VM on the local machine in case the executable is being launched from the remote binary server.

Since we leverage the Windows supported *Common Internet File System (CIFS)* sharing, redirecting the access is straightforward. When a process accesses one of its libraries using a local path, say "*C:\Program Files\abc.dll*", the FVM redirects the access to the binary server by renaming the system call argument to the path "[\\Binserv\C\Program Files\abc.dll](#)" (UNC naming). BinServ is the name of the binary server. This name is usually configured as a registry key on the binary server using a client tool. In addition to application binaries, FVM can also redirect loading of Windows system DLL's to the binary server by intercepting system calls accessing memory-mapped DLL images. When the system DLLs are loaded from the binary server, we require the end user machines to run the same OS kernel as the shared binary server.

The registry redirection logic is implemented entirely in the user space (unlike file redirection logic described above). The above technique of renaming the system call argument (file name) to perform file redirection doesn't work with registry related system

calls. In the current binary server prototype, we leverage two Winnt API's *RegLoadKey* and *RegStoreKey* to facilitate registry redirection. We export registry entries under the "HKLM\Software" (using *RegLoadKey* API) on the binary server and load these registry entries into the VM (using *RegStoreKey* API) that is set up to run shared binaries when the VM is first created. This implies that the client has a copy of all the registry entries associated with the applications installed on the binary server, and the registry access from the shared binary VM is redirected to its local copy. We also periodically synchronize the shared binary VM's local registry copy with those on the central server to reflect newly installed applications.

In addition to files and registries, applications can install environment variables on the binary server. To allow an application running in a shared binary VM to access its environment variables that are set up at the installation time, the shared binary VM retrieves all the environment variables stored on the binary server and merges them with environment variables of the local environment.

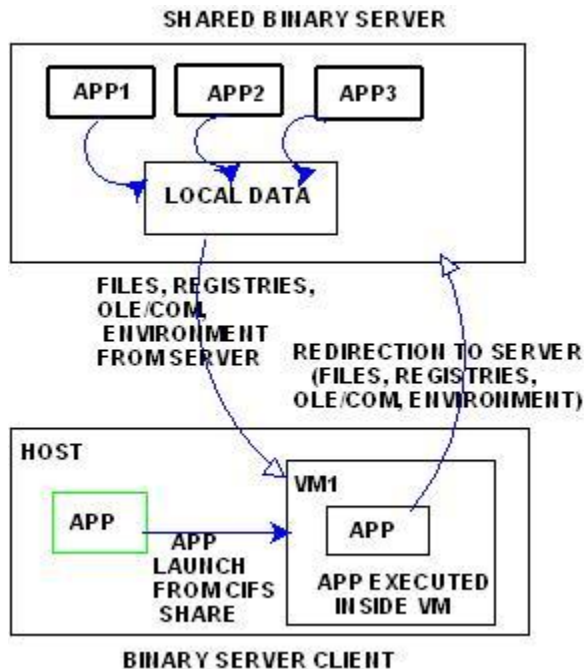


Figure 4.2: Binary server architecture.

COM (*Component Object Model*) is a platform independent, distributed and object-oriented system for creating binary software components that can interact. The COM framework provides a standard way for applications (.exe files) or libraries (.dll files) to make their functionality available to COM compliant applications. The shared binary server architecture also redirects accesses to the COM components that shared binary server depend upon. There are typically 2 types of COM components. One is called *In-Process* objects, which are usually implemented as DLL's that an application can load into its address space. Redirecting access to In-Process COM objects is same as redirecting accesses to DLL's. The other is Out-of-process COM component, usually

implemented as stand-alone exe's, which runs as a separate process. The Out-of-process COM components allow multiple client processes to connect to it. So, when an application program running in a shared binary server VM accesses an out-of-proc COM object, it should load the COM object from the remote binary server into the application regardless of any locally installed COM objects.

COM objects are identified by CLSID's (Class Identifiers), a globally unique identifier. To implement the above-described behavior for Out-of-process COM objects, we assign a new CLSID to each COM object accessed by the applications running inside a shared binary VM. We perform the necessary mapping between the old CLSID and the new CLSID during each COM object access. This mapping is maintained inside FVM. This forces the system to load COM objects from the binary server into the application process.

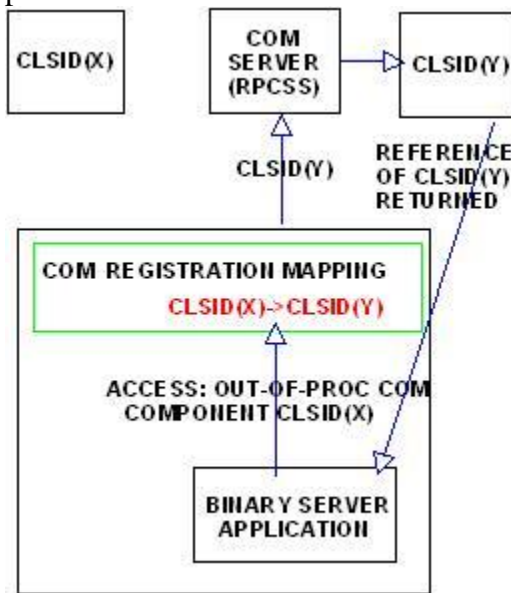


Figure 4.3: COM redirection technique used in Shared Binary server implementation.

By redirecting file, registry and OLE/COM access to a remote binary server in the FVM layer, the binary server VM allows users to use Windows applications without installing them on the local machine. We have tested a few applications, such as LeapFtp, Winamp and Microsoft Word, which cannot be started through the file-sharing interface unless running in the binary server VM.

<p>FILE REDIRECTION</p>	<ol style="list-style-type: none"> 1. "\\Program Files" 2. "\\Documents and Settings\\All Users" 3. "\\Windows" 4. "\\Winnt"
<p>REGISTRY REDIRECTION</p>	<ol style="list-style-type: none"> 1. \\HKLM\\Machine\\Software

Table 4.1: Shared Binary Server redirection logic

4.4 Evaluation

Correctness

We tested out the shared binary server implementation using a bunch of standard Windows applications (listed in table 4.1). We configured a shared binary server with the applications installed and shared the installation and other configuration directories. The client did not have any of the applications installed locally. When the executable from the server was launched from the client machine, the redirection strategy ensured that the application was launched successfully on the client.

Some of the Operating system dependent known DLL's (like the kernel32.dll) are not streamed to the client from the shared binary server. We do reuse some of the system DLLs of the client. This warrants that the operating environment on the client and the server be the same. Since the shared binary server architecture is built on top of FVM (an OS level virtualization technique), any application modifying the running kernel (by direct access) or loading a kernel module is not supported. Shared binary server architecture works only for user land applications.

The suggested in the previous section, the current redirection strategy of shared binary server is static. The redirection logic we use is based on heuristics. We notice that, in general, application installations usually update only a few directories (installation directory typically being C:\Prog~ files, configuration directories being C:\Docume~settings\\All users and system directory being C:\winnt). The application installation also modifies only fixed registry keys (HKLM\Software). Our redirection logic filters these directories/registry key entries to perform redirection. Testing of variety of Windows applications suggest that this heuristic is reasonably effective.

Performance Evaluation

We evaluate the performance of FVM based shared binary server architecture by measuring the startup time of six interactive Windows applications in the following configurations.

- Local installation and Execution (LIE): Applications are installed and executed on the end user machines

- Shared Binary Service with Local Data (SBSLD): Applications are installed on a central server and executed on an end user machine with input/output files stored locally.
- Shared Binary Service with Remote Data (SBSRD): Applications are installed on a central server and executed on an end user machine with input/output files also stored on the central server.
- Thin Client Computing (TCC): Applications are installed and executed on a central server, with execution results displayed on an end user machine through a Windows Terminal Service (WTS) session. Input/Output files are stored in the central server as well.

We use a test harness program to launch an application under test using the *CreateProcess()* Win32 API, and monitors the application's initialization using the *WaitforInputIdle()* API. The start-up time of a test application corresponds to the elapsed time between the moments when these two API calls return. To measure the initialization time of a WTS session, we run a terminal service client ActiveX control program on an end user machine. When a WTS session is successfully established, this program receives a connected event. The time between this event and the time when the program first contacts the terminal server is the WTS session's initialization time. We use two machines in this experiment. The client machine was an Intel Pentium-4 2.4GHz machine with 1GB memory running Windows 2000 server and the shared binary server machine was an Intel Pentium-4 2.4GHz machine with 256 MB memory running Windows 2000 server.

Caching

We tested out the effectiveness of caching the application configuration files and registries locally. The following is the summary of the test setup.

- (1) Inside the FVM, whenever we do the rename of the file/registry entries to the remote binary server, we cache it locally.
- (2) Any subsequent access to a resource is first looked up in the temporary directory (from (1)). We redirect the access to the binary server only when the file is not present locally
- (3) The registry entries are anyways cached on the client during the binary server setup time as described above. Figure 4.5 shows the startup comparison among various Windows applications.

Note that configurations SBCLD and SBCRD are equivalent to SBSLD and SBSRD respectively with local caching.

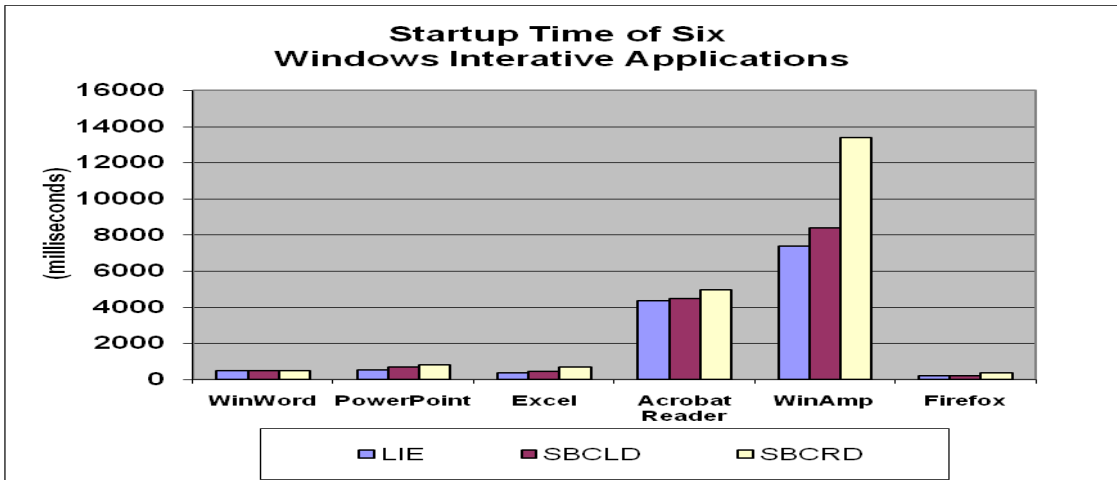


Figure 4.4 Start-up time of six interactive applications under 3 different test configurations.

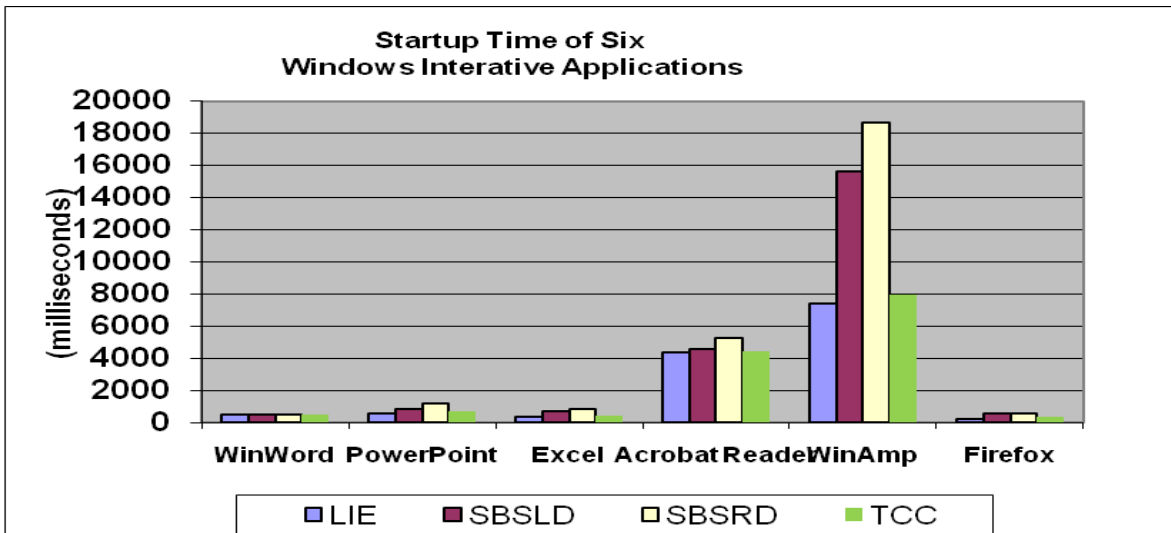
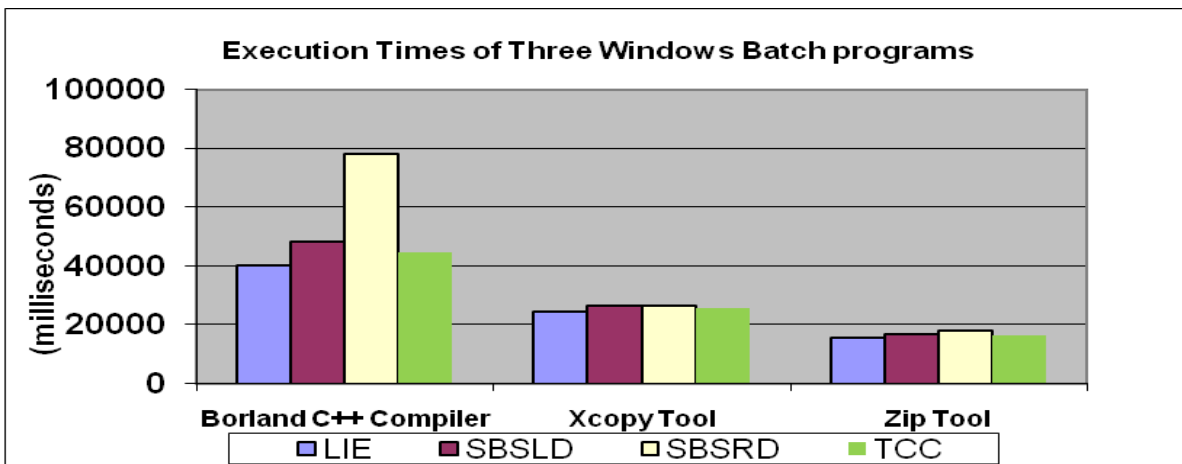


Figure 4.5 Start-up time of six interactive applications under 4 different test configurations.

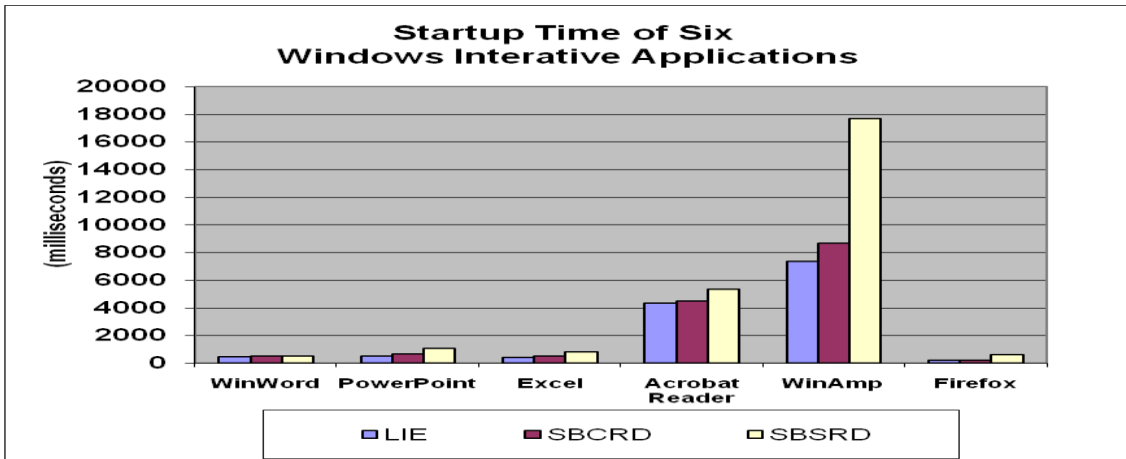


Figure 4.6 The startup time of six interactive applications and the execution times under 3 different configurations.

Analysis

Figure 4.5 shows the start-up time comparison among these four configurations for the six test applications. The start-up overheads of the four configurations tested are in the following order: SBSRD > SBSLD > TCC > LIE. In general, the amount of file/registry access over the network determines the magnitude of the initialization overhead. For LIE and TCC, the amount of file/registry access over the network is zero, and therefore their start-up times are smaller. Because TCC incurs an additional fixed cost (around 40 msec) of initializing a WTS session, its start-up time is longer than LIE's. Both SBSLD and SBSRD require access to the remote server for configuration files, registry entries, DLLs and COM/OLE components, and therefore incur a substantially higher start-up overhead. In the case of SBSRD, it incurs network access overhead even for input/output files and therefore takes longer to start-up than SBSLD.

The overhead order among the four configurations is different for WinWord. In this case, the start-up times of the four configurations are pretty close to each other, LIE: 463 msec, SBSLD: 479 msec, SBSRD: 494 msec and TCC: 517 msec. The reason that TCC exceeds SBSLD and SBSRD is because of its additional 40-msec WTS session initialization cost. The start-up time of WinAmp is much higher than that of other test applications because the number of file access redirections is much higher (214) than others, as shown in Table 5.2. In other words, WinAmp requires access to 214 executable or configuration files, which are fetched from the binary server at the initialization time.

Therefore, the execution of WinAmp using a binary server incurs a much larger start-up overhead than others. The large difference in SBSLD's and SBSRD's start-up times' for WinAmp arises from the large input file used in the test of WinAmp, which is 5.2Mbytes in size. In addition to interactive applications, we also measured the execution time of three batch programs in the above four configurations. The batch program's execution time in the four configurations follows the same order as the start-up time measurement for interactive applications, as shown in Figure 4.2

Application	Registry Redirections	File redirections
WinWord	996	62
Power Point	487	30
Excel	339	17
Acrobat Reader	152	55
WinAmp	397	214
Firefox	159	16

Table 4.2: The number of registry and file access redirected to the binary server during the initialization time for six interactive Windows applications.

Figure 4.4 shows similar testing scenario with caching configured. We cache the configuration files, registries, OLE/COM components locally in a temporary directory as described above. Clearly, the performance of shared binary server with caching and local data matches up with the performance of the local installation and execution results. The discrepancy in the performance could be attributed to the redirection/renaming overhead involved for each of the file/registry access. The amount of registry/file redirection is pretty prominent in case of WinAmp as seen from table 4.1. The input file size used for testing WinAmp was an mp3 file of size 5.2 Mega bytes which is the reason for the prominent discrepancy between SBCLD and SBCRD in table 4.4.

Chapter 5

Conclusion and Future Work

In this thesis, we describe some of the extensions to core FVM and an application of FVM (shared binary server). FVM prototype, with the described extensions, went open source on source-forge [13] last month.

Featherweight Virtual Machine (FVM) allows a single machine to host multiple isolated execution environments on a single Windows kernel. By name space virtualization and copy-on-write, FVM enables multiple VMs to efficiently share resources without interfering with one another. FVM's minimal VM startup/shutdown cost and large scalability make it an excellent platform for building applications that require frequent invocation and termination of “dispensable” VMs.

We have successfully leveraged FVM to implement a shared binary server prototype that exhibits moderate performance overhead. One of the future works, leveraging both the FVM framework as well as the shared binary server idea, is to build a distributed DOFS architecture to protect confidential files on the file server against information theft. The idea is to run file viewing/editing programs in a VM, which redirects all the operations back to a central server. Confidential files are encrypted and decrypted on the fly as they transmit over the network to ensure end user machines never have plain text file content.

The COM virtualization technique we used for shared binary service application is superseded by the FVM COM server (rpcss) virtualization. Virtualizing rpcss effectively ensures that there is one instance of the COM server per VM. Hence, the renaming technique we used in shared binary server becomes redundant (rpc server virtualization was implemented post FVM 1.0). Virtualizing rpc server was a bit tricky (brief synopsis described in section 3.3) but most of the issues pertaining to out-of-proc COM components falls into place. In hindsight, probably, we could have included FVM COM virtualization as part of design of shared binary server instead of designing an alternative solution for it.

File virtualization method in shared binary server involves renaming all local file access (based on the directories listed in table 4.1) to a CIFS network shared file path. However, this technique doesn't work for registry redirections as there is no interface in the Windows Kernel to redirect local registry access to remote registry. We currently fetch the remote registry *hive* (Table 4.1) and load it locally on the client on a new hive. Local registry accesses are mapped onto this newly created hive. The problem with caching registry keys on the client is any software updates on the server (which might potentially add/change registry key entries) is not immediately reflected on the client (not until the binary server client is initialized again). A probable solution would be to create

and rely on a user land daemon to perform registry redirections (Windows provides documented API's for remote registry accesses in the user land). One way to implement this could be to completely move the registry virtualization code to user land. This, however, would make the design feeble as subverting a user land interception is easier. Another solution would be to continue doing the registry virtualization inside the kernel, but create and rely on the user land daemon to fetch/set the remote registry data. This method is more robust but there is a performance hit as each registry access involves a shift from kernel to user mode to fetch the data.

To distinguish between accesses to local input/output data and those to application-specific configuration data, the binary server could be enhanced to monitor an application's installation process and record its configuration files, DLLs and registry settings. The resulted application profile can then be sent to the client VM as redirection criteria. This could be a rationale enhancement over the simple redirection criterion we use in the current prototype. FVM logging framework could be leveraged to do such profiling. FVM logging framework, in its current forms, logs the system call activity of all applications running inside the FVM. We could leverage this framework to automatically come up with a application specific selective virtualization profiles.

Bibliography

- [1] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In Proceedings of the 3rd USENIX Windows NT Symposium, July 1999.
- [2] Citrix Ardence. Software-streaming platform.
<http://www.ardence.com/enterprise/products.aspx?id=56>.
- [3] Microsoft Corporation. Technical overview of windows server 2003 terminal services. <http://download.microsoft.com/download/2/8/1/281f4d94-ee89-4b21-9f9e-9accef44a743/TerminalServerOverview.doc>, January 2005.
- [4] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual Appliances for Deploying and Maintaining Software. In *Proceedings of 17th Large Installation Systems Administration Conference*, October 2003.
- [5] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The Collective: A Cache-Based System Management Architecture. In *Proceedings of the 2nd Symposium*
- [6] Microsoft. Soft Grid Application Virtualization.
<http://www.microsoft.com/systemcenter/softgrid/default.mspx>.
- [7] AppStream. AppStream Technology Overview.
<http://www.appstream.com/products-technology.html>.
- [8] Bowen Alpern, Joshua Auerbach, Vasanth Bala, Thomas Frauenhofer, Todd Mummert, and Michael Pigott. PDS: A Virtual Execution Environment for Software Deployment. In *Proceedings of the 1st International Conference on Virtual Execution Environments*, 2005.
- [9] Thinstall. Application Virtualization: A Technical Overview of the Thinstall Application Virtualization Platform.
<http://thinstall.com/assets/docs/ThinstallWPApplicVirtualization4a.pdf>.
- [10] Yang Yu, Hariharan Kolam Govindarajan, Lap-Chung Lam and Tzi-cker Chiueh, “Applications of a Feather-weight Virtual Machine”, to appear in Proceedings of the 2008 International Conference on Virtual Execution Environments (VEE’08), March 2008
- [11] Yang Yu, Fanglu Guo, Susanta Nanda, Lap chung Lam, and Tzi cker Chiueh. A Feather-weight virtual machine for windows applications. In Proceedings of the 2nd International Conference on Virtual Execution Environments, June 2006.
- [12] Ph.D. dissertation on “*Feather-Weight Virtual Machine*” by Yang Yu.
<http://www.ecsl.cs.sunysb.edu/tr/TR223.pdf>
- [13] Feather-Weight Virtual Machines on sourceforge.net
<http://sourceforge.net/projects/fvm-rni/>
- [14] Softricity (SoftGrid)
<http://www.microsoft.com/systemcenter/softgrid/default.mspx>
- [15] Install Shield – Installation development tool.
<http://www.installshield.com/>
- [16] Install Anywhere – Installation development tool
www.zerog.com/

[17] WISE – Installation development tool

www.wisesolutions.com

[18] Script Logic - Installation development tool

www.scriptlogic.com