Facilitating Inter-Application Interactions for OS-level Virtualization

Zhiyong Shan[†] shanzhiyong@ruc.edu.cn
Xin Wang[‡] xwang@ece.sunysb.edu
Tzi-cker Chiueh[‡*] chiueh@cs.sunysb.edu
Xiaofeng Meng[†] xfmeng@ruc.edu.cn

[†]Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), MOE
[‡]Stony Brook University
[*]Industrial Technology Research Institute

Abstract

OS-level virtualization generates a minimal start-up and run-time overhead on the host OS and thus suits applications that require both good isolation and high efficiency. However, multiple-member applications required for forming a system may need to occasionally communicate across this isolation barrier to cooperate with each other while they are separated in different VMs to isolate intrusion or fault. Such application scenarios are often critical to enterprise-class servers, HPC clusters and intrusion/fault-tolerant systems, etc. We make the first effort to support the inter-application interactions in an OS-level virtualization system without causing a significant compromise on VM isolation. We identify all interactive operations that impact inter-application interactions, including inter-process communications, application invocations, resource name transfers and application dependencies. We propose Shuttle, a novel approach for facilitating inter-application interactions within and across OS-level virtual machines. Our results demonstrate that Shuttle can correctly address all necessary inter-application interactions while providing good isolation capability to all sample applications on different versions of Windows OS.

Categories and Subject Descriptors
D.4.5 [**Operating Systems**]: Reliability; D.4.6 [**Operating Systems**]: Security and Protection

General Terms
Reliability, Security

Keywords
OS-Level Virtual Machines, Inter-application Interactions, Cross-VM Communications, Intrusion/Fault Isolation

1 Introduction

OS-level virtualization partitions the OS name space to form a number of separated Virtual Machines (VMs), i.e., containers. VMs on the same OS share a single OS kernel and the host environment, and each VM only preserves state changes within its local environment. Programs in a VM run as normal applications that directly use the host OS' system call interface and do not need to run on top of an intermediate hypervisor. Accordingly, such VMs have a minimal startup/shutdown cost, low resource requirement and high scalability. Thus OS-level virtualization is applicable for the applications that require both high performance and good isolation [23][28], including intrusion/fault toleration [6][28][29], server consolidation [19][27], high performance system [23][26], distributed hosting organizations like PlanetLab [5][23], as well as cloud computing in the future [3][23].

These system functions often involve a set of member applications. To isolate intrusion or fault, the member applications are distributed into different VMs. On the other hand, the member applications occasionally require inter-application interactions which are essential for their execution, thus communications across VM barriers are inevitable.

The challenge is how to correctly and accurately handle all necessary inter-application interactions while not significantly affecting the isolation effectiveness of virtualization. *Inter-application interactions* represent the operations between distinct applications, e.g., register, notify, request, reply, authenticate and launch. Depending on whether two involved applications are located in the same VM or the host space, inter-application interactions can be ascribed into three basic categories: cross-VM, intra-VM and intra-Host. Intra-Host interactions represent the original inter-application interactions in the host environment and thus do not depend on virtualization technology. Cross-VM interactions need to penetrate the VM boundaries which are normally forbidden by the virtualization mechanism. Cross-VM interactions can be further divided into two sub categories, VM-Host where two involved applications run inside a VM and in the host environment respectively, and VM-VM where two applications reside in two different VMs.

The VM-Host interactions apply to all forms of OS-level virtualization technologies due to their nature. As OS-level VMs co-located on a host share a single OS kernel and the host environment, in order to access the essential system services (e.g., authentication, application initialization) and resources (e.g., Windows registry) in the host environment, an application in a VM has to interact across the VM boundary with applications in the host environment. For example, on Linux VServer [13], an application running in a VM has to authenticate itself to processes *sshd* and *getty*, which are run in the host environment. Likewise, on FVM [28], an application in a VM needs to authenticate itself to a host-resident process *lsass*. Since the authentication operation violates the isolation principle, the virtualization mechanisms drop the request, which leads the application in a VM to be suspended.

The VM-VM interactions are needed for cooperating applications to interact with each other to achieve certain goals. For example, a high performance computing software may distribute a group of cooperative programs into different VMs [9][27] in order to isolate intrusion/fault/performance/function [28] or concurrently foster multiple instances of the same program in a single OS. However, the virtualization mechanism will prevent the required VM-VM conversations from being carried across VM boundaries. Applications running in separate VMs on a single OS

VEE'12, March 3–4, 2012, London, England, UK.

might choose to exchange data using network communications to avoid penetrating through VM boundaries, but this would dramatically degrade the performance.

The intra-VM interactions are needed when two involved applications stay in the same VM and thus do not need to penetrate VM boundaries. However, improper startup sequence of the involved applications will cause the communications among them to fail. Accordingly, to prevent these failures, various types of OS-level virtualization platforms, e.g., Solaris Zones [20], Linux VServer [23] and FVM [28], invoke many unnecessary daemons or Windows services when booting and running a VM. This heavily slows down the booting procedure and increases the runtime overhead of a VM. As a result, it reduces the scalability of the OS-level virtualization technology, the main strength against the hardware-level virtualization [23][28]. Therefore, properly handling intra-VM interactions is necessary for improving the scalability of OS-level virtual machines.

In short, cross-VM and intra-VM interactions are often needed no matter what OS-level virtualization technology is used. Without a proper treatment, they will affect the running of the interaction-dependent applications and the scalability of OS-level VMs.

Moreover, as many interaction-dependent applications are important, it is necessary to enable their interactions efficiently. First, some of interaction-dependent applications are fundamental to the running of numerous other applications, for example, *RPCSS*, the RPC binding service; *PlugPlay*, the plug and play service; *NetDDE*, the distributed clipboard service. Second, some of interaction-dependent applications are critical to business organizations, such as web servers, database servers, high performance grid applications, transaction processing applications and enterprise-class applications. When exploiting the OS-level virtualization technology to consolidate servers or tolerate intrusion/fault, these interaction-dependent applications are deployed inside different VMs and need to communicate with other applications across the VM boundary or within a VM.

However, accurately identifying all possible inter-application interactions is not easy, as many applications are complex and their actual interactions vary across a wide range and are undocumented. Particularly on a commercial OS, as the OS and applications are close-sourced and most implementation details are kept confidential, identifying inter-application interactions poses a great challenge.

As far as we know, there is no scheme designed to systematically handle inter-application interactions in the literature. Existing papers concerning OS-level virtualization mostly focus on the general architecture of a specific type of OS-level virtualization [10][20][23][24][28], or exploit OS-level virtualization to consolidate servers[17][20], isolate intrusions [28][29] or build high performance systems [23], but never give a deep insight on the inter-application interactions. A few projects investigate how to improve cross-VM communications for hardware-level virtualization [6][9][27][30]. However, the cross-VM communications between hardware-level VMs only involve TCP/UDP-based network communications instead of inter-process communications that often occur across OS-level VMs, e.g., named pipe and event.

In this paper, we first investigate the interactive operations that affect the inter-application interactions using tracing and reverse-engineering skills. Based on the studies, we ascribe the interactive operations into four types: inter-process communications, application invocations, resource name transfers, and application dependencies. To address these issues, we design Shuttle, a novel approach that aims to facilitate all categories of inter-application interactions by intelligently handling four types of interactive operations while not leading to significant compromise of the isolation requirement of VMs. To demonstrate its effectiveness, we implemented Shuttle under the framework of Feather-weight Virtual Machine (FVM) [28] on different Windows platforms. The evaluations demonstrate that Shuttle can successfully support all tested Windows applications that depend on inter-application interactions with little impact on the VM isolation.

Table 1. Interactive operations affect certain inter-application interactions. Checks indicate the affected interactions.

Interactive Operations	Interaction Categories			
	Cross-VM		Intra-VM	Intra-Host
	VM-Host	VM-VM		
Inter-process communications	√	√	×	×
Application invocations	√	×	×	×
Name transfers	√	√	×	×
Application dependencies	×	×	√	×

Shuttle is the first approach to handle inter-application interactions for OS-level virtualization. With this approach enforced, multiple instances of the RPCSS, Dcomlaunch, SQL Server and IIS can concurrently run on top of a single Windows OS, which are believed almost impossible previously [28]. As the approach depends little on a specific operating system or OS-level virtualization technology, we believe it can also be generalized and applicable to different types of OS or OS-level virtualization technology.

The paper is structured as follows. The next section introduces the results of our studies on inter-application interactions. The approaches to handling cross-VM and intra-VM inter-application interactions are described in Section 3 and 4 respectively. Section 5 presents the implementation of Shuttle on FVM. Section 6 evaluates the prototype with a group of Windows applications on different versions of Windows OS. Section 7 discusses the applicability of Shuttle techniques to virtualization on other types of OS. Section 8 compares this research with other related efforts in the literature. Section 9 concludes the work.

2 Study on Inter-Application Interactions

As most of the interactive applications are close-sourced, their internal implementation details, for example, the internal logic, kernel objects created, registry entries accessed, etc., are rarely documented and open to the public in the literature. In order to investigate the exact interactive operations, we have spent several months to dynamically trace and analyze their behaviors, and statically reverse-engineer their binaries. Concretely, we take three investigation methods. First, we trace the kernel-level and Windows API-level calls that an application invokes at run time in order to determine the set of resources an application accesses. Second, we use the tool ProcessExplorer[15] to find out the inter-application communication objects an application uses to interact with other applications. Last, we disassemble the application's binary code to identify all hard-coded resource names and API function calls that transfer the hard-coded resource names.

We conclude that there are basically four types of interactive operations affecting inter-application interactions. Without a proper treatment of these operations, the interactive applications would fail or behave abnormally. These operations are as follows:

- Inter-process communications carried between two applications through IPC (Inter-Process Communication)

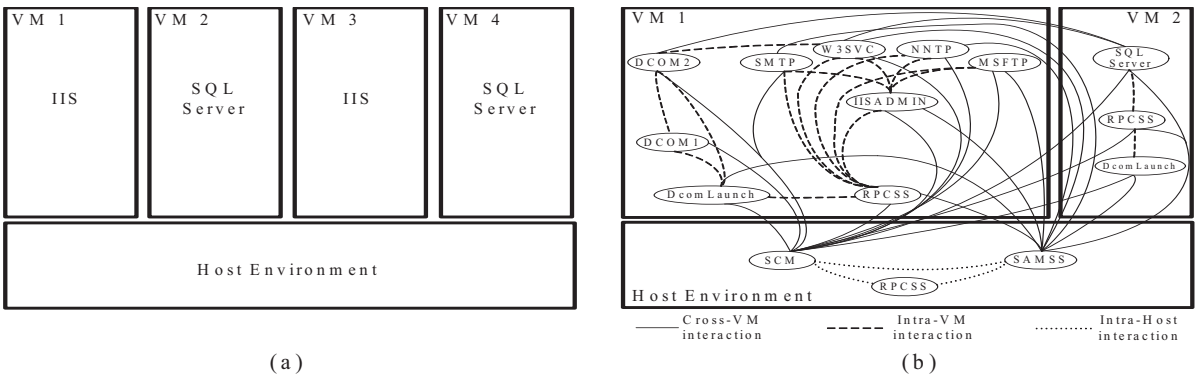


Figure 1. An illustration of inter-application interactions, running two groups of enterprise applications in different VMs on a single OS.

objects. Such operations include register, authenticate, request, reply, notify, exchanging data, etc.

- Application invocations where an application in the host starts other applications in a VM, which include daemons, Windows services, COM servers, etc.
- Name transfers needed for transferring resource names among applications running in different VMs. These names are hard-coded in application binaries and thus may escape the renaming mechanism of OS-level virtualization.
- Application dependencies when the running of an application (the *dependent application*) depends on the running of another application (the *master application*) in the same VM. The master application should run prior to the dependent application.

Table 1 summarizes our investigation results which associate the failures of inter-application interactions with the types of interactive operations. From the table, the former three types of operations might cause cross-VM interactions to fail while the last type might cause intra-VM interactions to fail. Further explanations of these results are presented in Section 3 and 4.

In order to better present the problems and our solutions, we perform a case study on a classical enterprise application in our lab using FVM [28]. We deploy two IIS servers and two SQL Servers in four distinct VMs on a single OS, as shown in Figure 1 (a). The two pairs of IIS and SQL Servers form two websites. Only with such a deployment, two instances of IIS or SQL Server can be separated without interfering with each other though they share a single OS kernel. Moreover, since each application is contained in a separate VM, the system constructed this way has the capability of intrusion/fault toleration. Similar deployments also can be found from Solaris Zones [20], OpenVZ [17] and Fido on Xen [6]. An alternative deployment scheme might place the two pairs of IIS and SQL servers into two VMs respectively. However, this setup not only can not completely avoid carrying out inter-application interactions cross-VM and intra-VM but also is not able to provide isolation as good as the former scheme.

Figure 1 (b) illustrates the detailed inter-application interactions across and within OS-level virtual machines, which only includes VM 1, VM 2 and the host. RPCSS is a fundamental Windows service on the Windows platform that provides RPC/COM/DCOM functions to other Windows services and applications, and is duplicated in each VM. Windows services are long-running programs that remain active without interacting with users, like daemons in a UNIX-style OS. Generally, there are about 100 services on Windows XP and nearly half of them depend on RPCSS. An IIS server consists of five Windows services: W3SVC service for web server, MSFTPSVC service for FTP server, SMTPSVC service for SMTP server, NNTPSVC service for network news server, as well as IISADMIN service for the management of IIS. DCOM1 and DCOM2 are a pair of DCOM servers started by DcomLaunch, which act as main and backup DCOM servers respectively. In Figure 1 (b), various specific inter-application interactions (e.g., register, notify, request, reply, launch and authenticate) among applications are represented by lines among them, which include cross-VM, intra-VM and intra-host interactions. The interactions among the applications are observed as a result of our efforts in tracing the process and performing reverse-engineering. However, due to these complex and hidden inter-application interactions, until now there is no public record showing a successful approach to making RPCSS or IIS run inside a VM.

3 Facilitating Cross-VM Interactions

3.1 Inter-Process Communications

According to the isolation principle of virtualization, the cross-VM inter-process communications should be strictly blocked although some applications may require interactions between each other. However, exceptions should be given to some essential cross-VM inter-process communications as shown in Figure 1 by solid lines. Hence, a carefully designed mechanism is required to facilitate these communications. To minimize the affection on VM isolation, the design should follow a principle: least penetration, which only allows least essential cross-VM communications.

3.1.1 Analyzing Cross-VM Inter-Process Communications

Corresponding to the Table 1, two categories of necessary cross-VM inter-process communications should not be blocked. One is the VM-Host communications between applications in a VM and the host environment, which possibly affect the VM-Host interactions. Such communications are often utilized by an application to get necessary services from core-applications, e.g., authentication and registration. The *core-applications* are the ones that provide system critical services to other applications, e.g., the Service Control Manager (SCM) on Windows, “launchd” on Mac OS and the “klogd”on Linux. They are actually the extensions of the OS kernel and closely tied with the kernel. They can not be duplicated in every VM and should stay in the host environment in order to be available to all VMs. Consequently this type of cross-VM communication is inevitable whenever an application inside a VM requests a system critical service.

The other is the VM-VM communications between applications in different VMs, which possibly affect the VM-VM interactions. In order to provide fault or intrusion isolation for individual applications, member applications belonging to the same system need to be placed in separate VMs [6][9][11][27][30]. Thus, the communications among these applications have to be carried across VM boundaries as exceptions to the basic isolation principle. For instance, storage systems (e.g., NetApp and EMC) may have a group of cooperative programs running in different VMs that need to communicate with each other. Similarly, a graphics rendering application in one VM may need to communicate with a display engine in another VM. Even routine inter-VM communications, such as file transfers or heartbeat messages may need to be performed frequently across the VM border.

Some of the VM-VM inter-process communications can be replaced by network communications, for example, using TCP/UDP communications to substitute named pipes. However, this will lead to a significant performance penalty [27] as the communication data need to go through the whole network stack twice in the same OS kernel. Therefore, facilitating VM-VM inter-process communications is indispensable not only for the successful running but also for better performance of cooperative programs.

Both categories of cross-VM inter-process communications are achieved via accessing Inter-Process Communications (IPC) objects, which have various types in an OS. For example, IPC objects in Windows include primitive ones (such as mutexes, events, timers, semaphores, and LPC) and higher-level ones (such as RPC and DCOM). Moreover, most actual IPCs between applications are undocumented and dynamic. Hence, it is difficult to thoroughly discover all the IPCs invoked by a running process, and decide which IPC should be confined within a VM and which IPC should not.

For the interaction-dependent applications that have well documented IPCs, one can manually give exceptions to permit the cross-VM communications to penetrate the boundaries of VMs. This is why existing OS-level virtualization technologies can successfully virtualize some interaction-dependent applications [23][28]. However, for the ones without documented IPCs, it is not feasible to manually identify all the cross-VM communications, especially for an ordinary user.

3.1.2 Handling Cross-VM Inter-Process Communications

In order to find a proper method to automatically handle cross-VM communications, we performed a study on the cross-VM IPC objects. The result shows that cross-VM IPC objects (e.g., named pipe, shared memory, mail slot, mutex, semaphore and socket) act at the server side of inter-application communications and hence must keep their names static to help the clients to locate them despite that the Id numbers of the objects are dynamic. For example, SQL Server prepares a pipe with static name "\Device\NamedPipe\sql\query" to wait for the connection request from local clients. In special situations, the name of an IPC object may partially change, i.e., with their name strings containing a number that changes over time. To address this issue, we can use a wildcard character * to stand for the variable number in the name string. In addition, a few special types of IPC objects might change names frequently, e.g., event. We can record the name of the receiver application rather than the name of the IPC object to help identify the IPC receiver.

Based on the study results, we devise our first Shuttle technique to leverage the names of cross-VM IPC objects to automate cross-VM communications. We employ *cross-VM endpoints* to point to the cross-VM IPC objects. A cross-VM endpoint in a VM is represented as $e=(n,i)$, where $e.n$ is the name of the corresponding cross-VM IPC object and $e.i$ is the Id of the VM containing the IPC object n. An IPC object is represented as $o=(n,t)$, where $o.n$ and $o.t$ represent the name and the type of the IPC object. The object type $o.t \in \{s,d\}$, where s means that the name of the object is static or contains a dynamic number, and d means that the name of the object is totally dynamic. An application is represented as $a=(n)$, where $a.n$ is the name of the application. A VM is represented as $v=\{i,O,E,A\}$, where $v.i$ is the Id, $v.O$ is the set of local IPC objects for intra-VM communications, $v.E$ is the set of endpoints for inter-VM communications, and $v.A$ is the set of applications of the VM. Accordingly, the logic to handle cross-VM communications can be formally described as follows:

$$
\begin{aligned}
&\textit{if}\ ((o_r.t = s \wedge (\exists o \in v.O, o_r.n = o.n)) \vee \\
&\quad (o_r.t = d \wedge (\exists a \in v.A, a_r.n = a.n))) \\
&\qquad \textit{DoIntraVMCommunication}(o_r); \\
&\textit{else if}\ ((o_r.t = s \wedge (\exists e \in v.E, o_r.n \approx e.n)) \vee \\
&\qquad (o_r.t = d \wedge (\exists e \in v.E, a_r.n \approx e.n))) \\
&\qquad\quad \textit{DoCrossVMCommunication}(e); \\
&\quad \textit{else} \\
&\qquad \textit{DenyCrossVMCommunication}(o_r);
\end{aligned}
$$

When an application in a VM v requests to access an IPC object o_r of application a_r, we first determine whether the required communication is intra-VM by checking the local IPC object list and the application set of the VM. Then, we search for the IPC object name or the application name in the cross-VM endpoint list of the VM. If an endpoint is found, we can quickly locate the corresponding cross-VM IPC object that serves the communication. We deny all other communication requests according to the isolation principle of the virtualization mechanism. The operator $\approx$ represents that the two involved names are two instances of the same IPC object. For example, according to the renaming rule in many OS level virtualization technologies [20][28], a port named *p* will be renamed in VM1 as *p-VM1* while in VM2 as *p-VM2*, thus we say $p\text{-}VM1 \approx p\text{-}VM2$.

3.1.3 Generating Cross-VM Endpoints

The challenge of implementing the technique is how to recognize all cross-VM IPC objects from thousands of candidate ones in an OS and form the cross-VM endpoints in a VM. Based on our studies, the cross-VM IPC objects provided by a type of application are mostly stable rather than changing over time in order to wait for connection requests from other applications. When the cross-VM IPC object name is stable, the corresponding endpoint uses the name of the IPC object, otherwise uses the name of the application.

As manually discovering the cross-VM IPC objects is almost impossible, we develop a tool to complete this task automatically by monitoring and recording cross-VM IPCs. For every type of application, we only need to test it once and can use the result in various application scenarios with different deployments. To prevent potential security issues (e.g., the occurrence of some unexpected cross-VM communications), we run the application only in a secure environment and right after the system and applications are installed. Moreover, to thoroughly discover all cross-VM IPC objects of the tested application, we tried various possible running conditions during the test. When all possible conditions were tested and there were no new cross-VM IPC objects appear, we stopped the test for the application.

Given the set of cross-VM IPC objects of different types of applications, the set of endpoints associated with an application is the union of the endpoints provided by all master applications that the application depends on. An application depends on a master

application as it has to access a cross-VM IPC object created by the master application. We say therefore that there is a cross-VM dependency between the dependent and master applications.

The cross-VM dependencies can be manually configured by users. The cross-VM dependencies between core-applications in the host and the applications in VMs are considered as default. Only the cross-VM dependencies between the cooperative applications running in different VMs require configuration. For example, if running a web server and a database server as a pair in two separate VMs on a single host, the administrator can configure a cross-VM dependency between the web server and database server.

However, when an application having a cross-VM IPC object runs multiple instances in different VMs, multiple instances of the cross-VM IPC object will confuse the applications which try to access one of them. Therefore, we introduce VM dependencies to address this issue. A *VM dependency* represents the cooperation between a dependent VM and a master VM. More specifically, an application running in the dependent VM will initiate a communication with the application running in the master VM. VM dependencies are generated by the following two rules:

First, when cooperating applications are deployed into different VMs, these VMs should have dependencies among them. For example, in Figure 1, as the IIS web servers and the SQL database servers are deployed in four VMs separately, the administrator should configure that VM 1 depends on VM 2 and VM 3 depends on VM 4 according to the existing cooperations. Second, the dependencies between the host and any VM are considered as default since applications in any VM require the services provided by the core-applications run in the host environment.

Taking into account the VM dependencies, the set of endpoints of a given application only contains the endpoints of the master applications running in the master VMs but not the ones running in other VMs. If a master application runs multiple instances in different VMs, only the endpoints of the instances in the master VMs are computed into the endpoint set rather than those from all VMs. Thus, the set of endpoints of a given VM is the union of the endpoints of all dependent applications running in the VM.

A question on our cross-VM communications technique is that the isolation offered by an OS-level VM might be compromised. There is actually a trade off between isolation and interaction. That is, virtual machines require isolation while interactive applications require cooperating with each other across VM boundaries. Hence, our technology follows the principle: least penetration, by only permitting the least necessary cross-VM communications. As presented above, this principle is followed by only allowing the communications between the applications that have predefined cross-VM dependencies and at the same time run in the VMs having predefined VM dependencies. This is in accordance with the basic principle of security protection: least privilege, which requires that every program of the system should operate using the least set of privileges necessary to complete the job [21].

3.2 Application Invocations

As shown in Table 1, cross-VM application invocations may cause some application failures. More specifically, some applications need to be cross-VM invocated by core-processes in the host environment, but the OS-level virtualization mechanism can not properly handle all of the cross-VM invocations, and thus such applications fail to be started inside VMs. Cross-VM invocations are inevitable as the core-applications responsible for launching such applications can not be virtualized, i.e., be duplicated in each VM. For example, all Windows services are started by SCM while SCM has to stay in the host as it is shared by all VMs and tightly

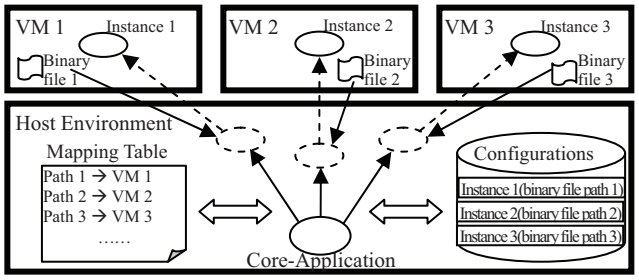


Figure 2. The mechanism for invocating applications cross-VM. It leverages a mapping table to help distribute multiple instances of the same application into multiple VMs.

related with the kernel. Linux, FreeBSD and Mac also have core-applications similar to SCM on Windows, e.g., init, getty and launchd, which are responsible for launching many daemons. Having tight relations with the kernel and providing shared services to many other applications, these core-applications are not allowed by the kernel to be duplicated in each VM. Hence, invocating applications cross-VM is also an issue for the OS-level VMs built on Linux, FreeBSD and Mac, e.g., Jails [10] and Linux VServer [23].

To handle cross-VM invocations, one can modify the application configuration database (e.g., Windows registry) or files to logically add a new instance of the application to be performed in a VM. Every time the core-application receives a request from a VM, it will fork a new process in the host, and then move the new process into the VM. However, as core-applications are not aware of the OS-level virtualization, it is difficult to decide which VM the new process should be moved into after the new process is generated. One can add extra information into the application configuration database to denote the VM that is requesting the new process. However, when multiple VMs simultaneously request to start the same application, we are still unable to correctly distribute multiple new processes into corresponding VMs.

To address this issue, we devise a novel mechanism that is illustrated in Figure 2. First, we prepare a distinct binary file for each application instance which is located in a distinct VM space, create a configuration entry containing the binary file path for each instance in the configuration database/file, and record the binary file path and the VM Id into the mapping table. Second, the core-application in the host environment starts a new process according to the corresponding configuration entry after receiving a startup request from a VM. Third, we intercept a new process and decide which VM the process should be placed into. The decision is made by searching the process' image file path in the mapping table so as to get the correct VM Id. Finally, we move the process from the host to the correct VM.

When starting an application from a VM the first time in response to a user request, Shuttle automatically prepares the binaries, registry entries and mapping table entry of the application. Specifically, when a user requests starting an application from a VM, Shuttle intercepts the request and checks whether the binaries and entries for the application running in the VM are prepared. If not, Shuttle prepares them and then forwards the request to the core application in the host to perform the start operation.

For example, an application CiSvc for indexing files has its image file at C:\WINNT\system32\cisvc.exe. To achieve cross-VM startup, we copy the file to the path C:\VMs\VM-Z\C\WINNT\system32\cisvc.exe that is within the space of VM Z, insert the path into a registry entry used to store the image path of

an application to be cross-VM started, and record the path and VM Id Z into the mapping table. When starting a process of CiSvc, we determine the VM that the new process should belong to by searching the process' image path in the mapping table and getting VM Id Z, and then move the process into VM Z.

A special case for cross-VM invocation is that a few applications are in the form of DLL (Dynamic Link Library). A DLL-based application runs as a thread inside a host process instead of an independent process as normal applications do. For example, the DcomLaunch service is a DLL-based application running as a thread inside a generic Windows host process called *svchost*. However, our mechanism still can handle this special type of applications. That is, for a DLL-based application, we record the host process' image file path in the mapping table to recognize VM Id rather than the application's DLL file path, as the thread of the application and its host process always live together within the same VM. However, as multiple host processes with the same image file often foster different DLL-based applications, it is difficult to differentiate these applications' VM Ids based only on the host process' image file path. We found that, to launch a DLL-based application, the host process has to use an exclusive parameter to indicate the running of the application. Accordingly, we attach the parameter at the end of the host process' image file path in the mapping table in order to recognize the application. Thus, we can determine the VM Id by searching both the new process' parameter and image file path in the mapping table, in order to place the new process into the correct VM.

For example, the DLL-based application DcomLaunch runs inside a svchost process with a parameter "-k dcomlaunch". To achieve cross-VM startup, we record not only the host process' image file path and VM Id but also the parameter into the mapping table. When starting a svchost process, we first obtain the Id of the VM that the new process should belong to by searching the process' image path and parameter in the mapping table, and then move the process into the corresponding VM.

3.3 Resource Name Transfers

When an application in a VM performs a cross-VM communication, it might need to transfer resource names (e.g., the application's name) to the receiver application in another VM or in the host environment. In some situations, such names are hard-coded and originated from the application's binary without being renamed according to the rules that OS-level virtualizations often employ [20][28]. When multiple instances of the same application run in different VMs simultaneously send a hard-coded name to the same receiver application, unexpected conflicts or errors will cause the instances to fail.

For example, the RPCSS service on Windows is such an application with the hard-coded application name (i.e., "RPCSS"). In its binary, two hard-coded RPCSS service name strings are used as input arguments by the service management function *OpenServiceW()*. When the RPCSS is required to start in VM Z, it calls *OpenServiceW()* to send a request to the SCM process running in the host environment, using the hard-coded RPCSS name as a parameter. *OpenServiceW()* in turn communicates across VM boundaries with the SCM through a named pipe NtControlPipe. The SCM then checks whether the requested service name is valid and without conflict. If another instance of RPCSS is running in the host environment or in another VM, SCM will refuse the open service request from VM Z since the hard-coded RPCSS name is already registered in SCM. As a result, the RPCSS process in the VM Z will fail.

The basic reason is that, the original developers did not anticipate that a program may be replicated with multiple application names. They simply hard-coded a fixed application name in the program codes and used it as an argument in subsequent calls to Win32 API functions, which send the name across VM boundary through an IPC channel.

On further investigation, we find hard-coded resource names in binary files on other OS platforms, e.g., Linux and FreeBSD. As OS-level virtualization technologies often rely on resource renaming to separate VM spaces [20][28], this issue is not exclusive on Windows OS or FVM. However, existing OS-level virtualization technologies are not aware of this issue, let alone providing any solution.

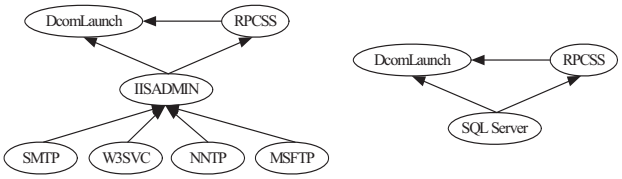
A possible solution to this issue is to intercept the related API functions and change the parameters that are originated from hard-coded names following the virtualization rules. However, one can not intercept all the related API functions which use hard-coded names. If an application in a VM invokes a function that uses a hard-coded name but is not intercepted, the solution will not be able to rename the parameters.

A better solution is to intercept only IPC related system calls that have a limited number in an OS. Once capturing a resource name in its original form in an inter-process communication, Shuttle changes it following the renaming rules, e.g., appending a VM Id to the name. However, filtering the contents of the inter-process communications to find the names might significantly slow down the system as the communications are often frequent and contain a fair amount of content. Fortunately, Shuttle can differentiate cross-VM from intra-VM inter-process communications by checking the cross-VM endpoints as presented in Section 3.1. Thus, we can focus on the cross-VM inter-process communications. As they represent a very small fraction of the entire inter-process communications in a system, monitoring cross-VM inter-process communications only imposes little overhead on the system. Moreover, transferring hard-coded name across VM boundaries can be only pursued through IPC objects. Therefore, the monitoring of IPC in Shuttle is general and can be extended to apply in other type of OS.

In the previous example, Shuttle monitors the pipe NtControlPipe and changes the application name string that the RPCSS writes into the pipe. Then SCM will permit the open service request with the service name containing the VM Id, as the changed name will no longer conflict with those used in other RPCSS instances running in the host environment or other VMs.

4 Facilitating Intra-VM Interactions

Although intra-VM communications between applications are permitted by OS-level virtualization technology, incorrect startup sequence of the applications would cause an intra-VM communication to fail. This is due to the dependencies among applications in the same VM as shown in the Table 1. In other words, as some applications (say *dependent applications*) depend on the running of other applications (say *master applications*) in



(a) In-VM Dependency Graph of VM 1 and 3 (b) In-VM Dependency Graph of VM 2 and 4

Figure 3. The intra-VM dependency graphs resulted from the enterprise application scenario in Figure 1.

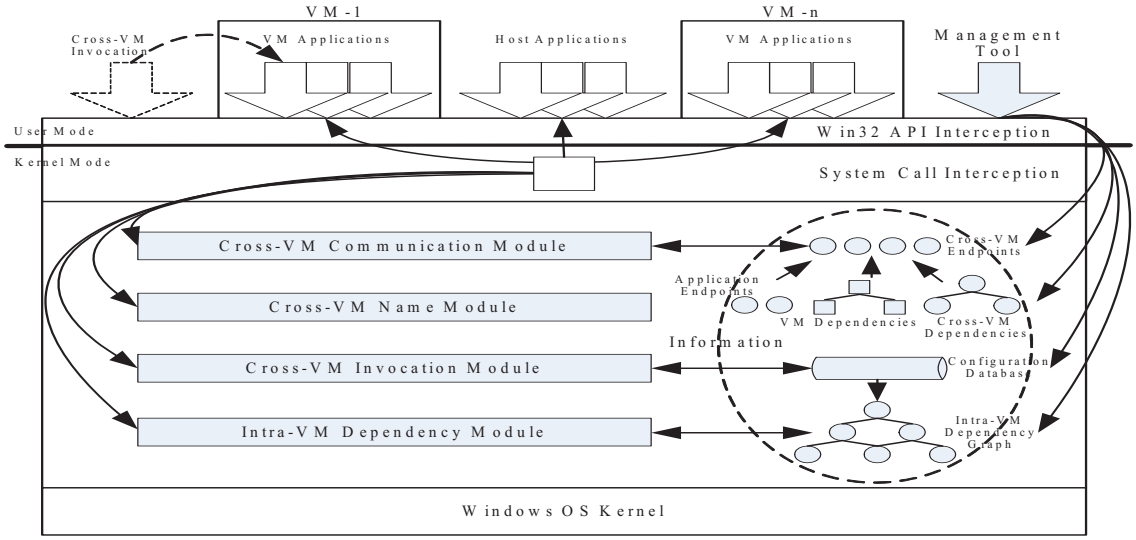


Figure 4. Shuttle architecture consists of four modules and six types of information in the kernel, as well as a management tool at the application level. The four modules handle the four types of interactive operations respectively based on the six types of information.

the same VM, the master applications should be started prior to the dependent ones.

This seems to be a standard OS design issue i.e., arranging for the startup scripts to start applications in the right order to satisfy their dependencies. However, to achieve VM scalability and thus increase the number of VMs that can be active concurrently on a single OS, OS-level virtualization has an extra high requirement on resolving this issue than that of a standard OS. That is, the solution of this issue should help to speed up the booting procedure or reduce system resource occupation of a VM to the minimum degree.

To avoid a possible application failure resulted from an intra-VM dependency, existing OS-level virtualization technologies [20][23][28] start many system applications that often serve as master applications when booting a VM through manual configuration, e.g., Linux daemons and Windows services. These applications then wait for the intra-VM communications initiated by other applications. This will significantly slow down the booting procedure of the VM and occupy extra system resources, e.g., CPU time and memory space. On the other hand, the system applications not being manually configured to start at the time of booting VM might later be required by some dependent applications, which thus causes unexpected failures of the dependent applications.

To address these issues, we propose a technique that starts master applications only on demand and stop master applications upon system idle, so that the master applications no longer slow down the VM booting procedure and unnecessarily occupy system resources at run time, as well as cause the failure of the dependent applications. This will improve the performance and scalability of OS-level virtualization technology.

As a core part, the technique uses an intra-VM dependency graph which can be represented as a set of vertices and a set of directed edges connecting the vertices. A vertex represents an application. An edge between two vertices represents a dependency between two applications. Based on the intra-VM dependency graphs, the start-on-demand and stop-on-idle policies can be specified as follows:

The start-on-demand policy starts a master application and increases it's referred count at the time a corresponding dependent application starts. The stop-on-idle policy decreases the referred count of a master application when a corresponding dependent application stops. Once the last dependent application of a master application stops, the policy stops the master application when it does not need to remain active.

The intra-VM dependency graph can be generated from application configuration files or database. For example, the registry entries `DependOnService` and `DependOnGroup` on Windows describe the dependencies between services and thus can be used to generate the graph. To automate the graph creation procedure, we develop a program to discover intra-VM dependencies among related applications by scanning the related entries in the Windows registry. As an example, Figure 3 (a) and (b) depict the intra-VM dependency graphs of the application scenario in Figure 1, which are generated by our program.

With these two policies enforced, an OS-level virtual machine does not need to start system applications at the booting time but only upon requests. Meanwhile, a VM does not need to foster the idle system applications. Accordingly, the overhead of booting and running an OS-level virtual machine can be significantly reduced, and the potential application failures incurred by the intra-VM dependencies can be avoided.

5 Shuttle Prototype

We have implemented Shuttle on FVM [28][29] to facilitate inter-application interactions within a VM or across VMs. Figure 4 shows the Shuttle architecture. All inter-application interactions, cross- and intra-VM, are captured by intercepting system calls in the kernel and Win32 APIs in Windows system libraries. Shuttle mainly intercepts system calls related to IPC, file, registry and process as they are often invoked for inter-application interactions. Some inter-application interactions that involve a few IPCs (e.g., message) and services can not be identified by intercepting system calls, we thus intercept the corresponding Win32 APIs.

To intercept the system calls, we modify the system call entry point in the *System Service Dispatch Table* (SSDT) within the

Table 2. Testing results of running interaction-dependent applications in VMs with the support of Shuttle. After addressing the interactive operations marked, all the applications can perform inside VMs and cooperate across VMs correctly.

Samples	Interactive Operations			
	Cross-VM Inter-Process Communications	Cross-VM Name Transfers	Cross-VM Application Invocations	Intra-VM Application Dependencies
RPCSS on Windows 2000	√	√	√	
IIS on Windows 2000	√	√	√	√
RPCSS on Windows XP	√	√	√	√
Dcomlaunch on Windows XP	√	√	√	
IIS on Windows XP	√	√	√	√
Mysql on Windows XP	√	√	√	√
Apache on Windows XP	√		√	
Tlntsvr on Windows XP	√		√	√
CiSvc on Windows XP	√		√	√
ImapiService on Windows XP	√	√	√	√
SQL Server on Windows XP	√	√	√	√
Ntsvcs on Windows XP	√	√	√	√
MS Word on Windows XP				√
MS PowerPoint on Windows XP				√
MS Excel on Windows XP				√
MS Office Assistant on XP				√
AutoCAD on Windows XP	√			√
Adobe installation on Windows XP	√			√
MS Office 2003 installation on XP	√			√
Regcmd installation on XP	√			√
StraceNT on Windows XP				√
ProcessMonitor on Windows XP				√

Table 5. Mapping table for cross-VM application invocations. Shuttle searches a new process' image file path within the table in order to get the corresponding VM Id, so that Shuttle can distribute the new process into the correct VM.

Binary file paths	VM Id
c:\fvms\VM1\C\WINNT\system32\inetsrv\inetinfo.exe (IIS)	1
c:\fvms\VM3\C\WINNT\system32\inetsrv\inetinfo.exe (IIS)	3
c:\fvms\VM2\C\PROGRA~1\MI6841~1\MSSQL\binn\sqlservr.exe (SQL Server)	2
c:\fvms\VM4\C\PROGRA~1\MI6841~1\MSSQL\binn\sqlservr.exe (SQL Server)	4
c:\fvms\VM1\C\WINNT\system32\svchost.exe -k rpcss (RPCSS)	1
c:\fvms\VM2\C\WINNT\system32\svchost.exe -k rpcss (RPCSS)	2
c:\fvms\VM3\C\WINNT\system32\svchost.exe -k rpcss (RPCSS)	3
c:\fvms\VM4\C\WINNT\system32\svchost.exe -k rpcss (RPCSS)	4
c:\fvms\VM1\C\WINNT \system32\svchost.exe -k DcomLaunch (DcomLaunch)	1
c:\fvms\VM2\C\WINNT \system32\svchost.exe -k DcomLaunch (DcomLaunch)	2
c:\fvms\VM3\C\WINNT \system32\svchost.exe -k DcomLaunch (DcomLaunch)	3
c:\fvms\VM4\C\WINNT \system32\svchost.exe -k DcomLaunch (DcomLaunch)	4

kernel. To intercept the Win32 APIs, we modify the library function entry point in the *Import Address Table* (IAT) of the application process. Shuttle adds and changes about 10k lines of code on FVM at both application level and kernel level.

The intercepted interaction requests are posted to four kernel modules to make decisions based on six types of information, which are presented as follows.

The Cross-VM Communication Module handles cross-VM inter-process communications based on the cross-VM endpoints.

Table 3. Endpoints for cross-VM inter-process communications. Every endpoint consists of the name of the cross-VM IPC object and the Id of the VM containing the object. The VM Ids listed on the right column indicate the VMs containing the corresponding endpoint.

Endpoints <IPC object name, Master VM Id>	VM Id
<\RPC Control\DNSResolver, 0>	1,2,3,4
<\RPC Control\ntsvcs, 0>	1,2,3,4
<\Device\NamedPipe\net\NtControlPipe*, 0>	1,2,3,4
<\Device\NamedPipe\ntsvcs, 0>	1,2,3,4
<\Device\NamedPipe\EVENTLOG, 0>	1,2,3,4
<\Device\NamedPipe\samr, 0>	1,2,3,4
<\Device\NamedPipe\sql\query, 2>	1
<\Device\NamedPipe\sql\query, 4>	3
<\BaseNamedObjects\DBWinMutex, 0>	1,2,3,4
<\BaseNamedObjects\RasPbFile, 0>	1,2,3,4
<\BaseNamedObjects\SHIMLIB_LOG_MUTEX, 0>	1,2,3,4
<\BaseNamedObjects\ShimCacheMutex, 0>	1,2,3,4
<\BaseNamedObjects__R_ 0000000000da_SMem__, 0>	1,2,3,4
<\BaseNamedObjects\DBWIN_BUFFER, 0>	1,2,3,4
<\BaseNamedObjects\ShimSharedMemory, 0>	1,2,3,4
<\BaseNamedObjects\ScmCreatedEvent, 0>	1,2,3,4
<\BaseNamedObjects\SvcctrlStartEvent_A3752DX, 0>	1,2,3,4
<\BaseNamedObjects\crypt32LogoffEvent, 0>	1,2,3,4
<\BaseNamedObjects\userenv: User Profile setup event, 0>	1,2,3,4
<\BaseNamedObjects\DINPUTWINMM, 0>	1,2,3,4
<\BaseNamedObjects\TESTMSSQLSERVER. 2>	1
<\BaseNamedObjects\TESTMSSQLSERVER. 4>	3
<\BaseNamedObjects\TESTMSSQLSERVER_MUTEX. 2>	1
<\BaseNamedObjects\TESTMSSQLSERVER_MUTEX. 4>	3
<\SECURITY\LSA_AUTHENTICATION_INITIALIZED, 0>	1,2,3,4

Table 4. Cross-VM names that are derived from hard-coded resource names and sent across VM boundaries by the corresponding applications, and in turn intercepted and renamed by Shuttle. Z indicates the Id of the VM that the corresponding applications live in.

Applications	Changed cross-VM names
RPCSS	RPCSS → RPCSS-vmZ
DcomLaunch	DcomLaunch → DcomLaunch-vmZ
IIS	W3SVC → W3SVC-vmZ
SQL Server	sqlservr → sqlservr-vmZ

The Cross-VM Name Module handles name transfers across VMs. It checks cross-VM IPCs to rename the hard-coded resource names derived from applications' binaries. The Cross-VM Invocation Module watches the processes that are cross-VM started and moves them into the corresponding VMs. Based on the intra-VM dependency graph, the Intra-VM Dependency Module suspends the processes to be started and starts their master applications first. It also monitors the processes to be stopped and requests to stop the master applications. For example, to stop a Windows service, Shuttle calls *ControlServiceEx()* with a control code SERVICE_CONTROL_STOP to request SCM to stop the service.

The six types of information used by the modules include cross-VM endpoints, cross-VM dependencies, VM dependencies, cross-VM endpoints of each type of application, intra-VM dependency graph and configuration database (i.e., Windows registry). We have five key data structures to represent the former

five types of information. The cross-VM endpoints are generated by the Cross-VM Communication Module and management tool based on the cross-VM endpoints of each type of application, cross-VM dependencies and VM dependencies, using the method presented in Section 3.1. The intra-VM dependency graph is generated by the Intra-VM Dependency Module and management tool by scanning certain registry entries, e.g., `DependOnService` and `DependOnGroup`.

Shuttle was implemented in two different versions of Windows OS, i.e., Windows 2000 and Windows XP, as FVM was implemented in these two versions. We believe with minor changes Shuttle also can be implemented in a newer version of Windows, e.g., Vista, because the architecture of Shuttle does not contain technical details about specific version of Windows OS.

6 Evaluation

In this section, we present details on the experimental evaluations of our Shuttle prototype which consists of three parts. First, we investigate the effectiveness of Shuttle approach using a number of interaction-dependent applications and an enterprise application as the case to study. Second, we test whether Shuttle will cause a significant degradation of the isolation capability of an OS-level virtualization system. Third, we evaluate the performance overhead of our Shuttle prototype. The test-bed used in the evaluation consists of two machines. Machine A contains a Pentium-4 2.8GHz CPU with 1GB memory running both Windows 2K and XP and machine B contains an Intel Core 2 Duo 2GHz CPU with 2GB memory running both Windows 2K and XP. We installed FVM and Shuttle on both machines.

6.1 Effectiveness

The objective of the Shuttle approach is to facilitate inter-application interactions cross- and intra-VM so that the applications depending on these interactions can perform inside VMs without failures. To demonstrate the effectiveness of the Shuttle approach, we have run 22 interaction-dependent applications in the Shuttle prototype. Many of the sample applications have failed to perform inside a VM without the support of Shuttle so far, e.g., RPCSS, IIS, SQL Server, Ntsvcs, AutoCAD, Adobe installation and MS Office assistant. To sufficiently test all potential interferences among separate instances of the same application, each sample application at least runs three instances simultaneously on a single host. One instance runs in the host environment and the other two run in two different VMs respectively. Table 2 shows the evaluation results. Each row presents an application and the interactive operations appeared when running the application inside a VM. With these interactive operations resolved by Shuttle, all samples tested can successfully run three instances simultaneously on a single host.

Moreover, the sample applications running in VMs behaved correctly, which was verified as follows. RPCSS, Dcomlaunch and Ntsvcs were verified by the successful running of many other tested samples that depend on these three applications. IIS, Apache, Mysql and SQL Server were verified by building and operating websites. Tlntsvr was verified by supporting a telnet site. The installation programs were verified by the successful messages appeared at the end of installation procedures. StraceNT, Regcmd, CiSvc and ProcessMonitor were verified by checking whether they can work properly. The remaining samples were verified by opening and editing corresponding type of files.

As a case study, we further set up an enterprise application scenario that runs two pairs of web server and database server in four VMs respectively on a single Windows XP OS, as presented in Section 2. The applications worked properly and quickly. In the VMs, we performed various operations including browsing web pages, submitting web forms filled, downloading and uploading files. To handle some operations, the web servers cross-VM accessed the backend database server deployed in another VM. By calling the function *DbgPrint()* in the kernel and analyzing the results displayed with a debugging tool DbgView, we obtained the information used by Shuttle to handle the four types of operations that impact application interactions, which are presented as follows.

For cross-VM inter-process communications, the endpoints of every VM are listed in Table 3. In the table, all VMs have almost the same endpoints, except that VM 1 and 3 have more endpoints to access corresponding SQL Server in VM 2 and 4. The * character following the IPC object NtControlPipe indicates an arbitrary number that changes over time, as introduced in Section 3.1. The IPC objects DBWinMutex and DBWIN_BUFFER are specially generated by the DbgView for printing messages. The cross-VM names intercepted and changed are listed in Table 4. For the cross-VM application invocations, Table 5 presents the mapping table that records the binary file paths of the applications to be cross-VM started and their corresponding VM Ids. The intra-VM dependency graphs of all VMs are depicted in Figure 3 (a) and (b).

Therefore, Shuttle can successfully support all four types of interactive operations summarized in Table 1, which in turn supports all necessary cross-VM interactions.

6.2 Isolation

In order to evaluate the impact on the isolation capability of FVM caused by Shuttle, we prepared 19 pairs of small programs to test all possible forms of cross- and intra-VM operations. Each pair of programs is responsible for testing one type of Windows OS object, which consists of a client and a server. The testing results are shown in Table 6. The Intra-VM column indicates the results of the accesses from a client to a server both of which are placed within the same VM. All Intra-VM accesses are permitted. The VM-VM and VM-Host columns indicate the access results from a VM to another VM and from a VM to the host, which mostly are refused. In other words, cross-VM operations are correctly blocked and thus the VM isolation is preserved.

There are two exceptions in the table. One is at the rows for socket, which allows connect and send operations to cross VM boundary, because network communications should be permitted. The other is at the rows for file, registry and device, which allows read operations to be carried from a VM to the host. This is the result of the copy-on-write policy of FVM which aims to avoid duplicating a huge volume of OS objects from the host to each VM environment. However, this should not affect the isolation since any write result is saved separately within the corresponding VM. In addition, for many types of objects, e.g., event, we test open operation instead of read and write, because read and write operations need object handles that are obtained by open operations.

We also tested 30 more individual applications. Every application can smoothly run three instances simultaneously in two VMs and the host environment separately. The applications are as follows: Google Chrome, Windows command prompt, Internet Explorer, Microsoft Clip Organizer, MS Outlook Express, MS Messenger, mIRC, Visual C++, Firefox, Adobe Reader, Bitcomet, Foxmail, Windows Media Player, Putty SSH client, WinRAR, Skype, Windows FTP client, Beyond Compare, Source Insight, Calculator, Utility Manager, Notepad, Minesweeper, Hearts, WebBench Client, Winamp, Internet Backgammon, Diffutils Installation, Registry Commander, fvmsetup.

Moreover, our former test in Section 6.1 also shows that Shuttle can provide enough isolation even when facilitating cross-

Table 6. Testing results of the isolation capability of FVM enforced with Shuttle, which can correctly block general cross-VM interactions including both VM-VM and VM-Host interactions. × and √ represent blocked and allowed operations, respectively.

Objects	Operations	VM-VM	VM-Host	Intra-VM	Objects	Operations	VM-VM	VM-Host	Intra-VM
File/ Directory	Read	×	√	√	Service	Create	×	×	√
	Write	×	×	√		Open	×	×	√
	Create	×	×	√		Start	×	×	√
FileMapping	Open	×	×	√	Window	Find	×	×	√
Data Copy	Send	×	×	√	Mutant	Open	×	×	√
Registry	Read	×	√	√	Semaphore	Create	×	×	√
	Write	×	×	√		Open	×	×	√
	Create	×	×	√	Named Pipe	Read	×	×	√
RPC	Send	×	×	√		Write	×	×	√
Device	Read	×	√	√	Clipboard	Get Data	×	×	√
	Write	×	×	√		Set Data	×	×	√
	Create	×	×	√	Socket	Bind	×	×	√
Process	Open	×	×	√		Connect	√	√	√
Mailslot	Open	×	×	√		Send	√	√	√
Event	Open	×	×	√	Message	Send	×	×	√
COM	Request	×	×	√	Port	Connect	×	×	√
Timer	Open	×	×	√		Request	×	×	√

Table 7. Interception overhead of system calls and Win32 API functions. Compared with old FVM without Shuttle, the new FVM enforced with Shuttle adds less than 13.8% extra CPU cycles for file-related system calls, less than 10.5% for IPC-related system calls, and less than 1.8% for service related API functions.

System calls and Win32 API functions		Native (CPU Cycles)	FVM (CPU Cycles)		Shuttle (CPU Cycles)		Overhead (%)
File	NtCreateFile	334,492	401,931	(20%)	403,297	(21%)	0.3%
	NtOpenFile	167,620	216,895	(29%)	218,435	(30%)	0.7%
	NtCreateNamedPipeFile	183,574	223,960	(21%)	240,481	(31%)	7.4%
	NtCreateMailslotFile	40,790	42,015	(3%)	47,807	(17%)	13.8%
IPC	NtOpenSemaphore	30,234	64,286	(113%)	69,840	(131%)	8.6%
	NtCreatePort	37,241	72,309	(94%)	79,901	(115%)	10.5%
	NtOpenSection	38,134	72,742	(91%)	80,234	(110%)	10.3%
Service	StartService	2,166,808,231	2,166,819,311	(<0.1%)	2,166,818,157	(<0.1%)	<0.1%
	RegisterServiceCtrlHandlerEx	2,865,374	2,865,609	(<0.1%)	2,865,481	(<0.1%)	<0.1%
	QueryServiceStatusEx	2,011,945	2,011,960	(<0.1%)	2,011,959	(<0.1%)	<0.1%
	CreateService	8,264,623	8,406,775	(1.7%)	8,264,803	(<0.1%)	-1.7%
	OpenService	5,490,443	5,490,570	(<0.1%)	5,589,401	(1.8%)	1.8%

VM interactions, as multiple instances of the same application can simultaneously perform inside different VMs and the host.

In short, the three serials of testing results above show that, Shuttle can successfully offer isolation functionality while providing necessary exceptions for essential cross-VM interactions. The major reason lies in our principle of least penetration. That is, we only allow the cross-VM communications with specific IPC object names between predefined applications run in predefined VMs. Therefore, the chance of compromising the isolation of a VM is reduced to the minimum level.

6.3 Performance

In this section, we show the impact of Shuttle on the performance of virtualized applications and virtual machines. As the performance overhead of Shuttle results mainly from executing additional instructions when intercepting system calls and API functions, we measure specifically the interception overhead of the corresponding system calls and API functions. First we disable the FVM layer, run a group of applications natively in the host environment, and count the average number of CPU cycles spent in each system call and API function with the *rtdsc* instruction. Then, we enable the FVM layer without Shuttle, run the same applications in a VM and take the same measurements. Finally, we enable the FVM layer with Shuttle, run the same applications and take the same measurements. Each of the reported numbers shown in Table 7 is an average of the results of 100 runs on machine A running Windows 2K.

Table 7 shows the interception overheads in terms of CPU cycles of a set of intercepted system calls and Win32 API

functions, including four file-related system calls, three IPC-related system calls and five service related API functions. The new FVM with Shuttle enforced takes up to 31% more CPU cycles than the native configuration for file-related system calls, up to 131% for IPC-related system calls and 1.8% for service related API functions. Although the per-system call overhead seems to be significant for IPC-related system calls, the end-to-end impact on the overall system performance is much smaller, because IPC-related system calls account for lower than 0.2% of all invoked system calls in our test applications. Moreover, compared with old FVM without Shuttle, the current implementation of the proposed Shuttle approach adds less than 13.8% extra CPU cycles for file-related system calls and less than 10.5% for IPC-related system calls. For most service related API functions, the new FVM is actually as fast as the old one or even slightly faster, because service names used in API functions do not need to be renamed when Shuttle is in place. From these results, we can conclude that the performance cost of Shuttle is quite acceptable.

7 Discussion

As Shuttle prototype is based on Windows OS, one might question whether it is applicable to the OS-level virtualization based on other OSes, e.g., Linux. First, the issues resolved by Shuttle should also appear in other OS-level virtualization systems, because they are irrelevant to specific OS or virtualization technique. As analyzed in Section 3.1.1, Section 3.2, Section 3.3 and Section 4, these issues are actually derived from the nature of OS-level virtualization or applications, and thus also occur in Linux-based virtualization systems. As some concrete examples, there is a list of programs that have problems with Linux-VServer [14]. Some of the problems correspond to the issues resolved in this paper, for example, the problem "OpenLDAP Startup" is caused by application dependency, "rndc" is caused by hard-coded name and "Links inside screen inside a V-Server" is caused by cross-VM invocation. The techniques in this paper should be useful when resolving these problems in Linux-VServer. Second, the shuttle solutions for these issues actually do not depend on specific OS, though we often use Windows OS as examples in order to present the techniques more clearly. Therefore, we believe Shuttle should be applicable to the OS-level virtualization systems on other OSes.

8 Related Work

As far as we know, there is no such a project that can successfully handle all types of inter-application interactions across and within OS-level VMs in the literature. There are three categories of projects close to our work.

The first category is OS-level virtualization projects that include FreeBSD Jail [10], Linux-VServer [23], Solaris Zones [20], Open VZ [24], FVM[28], Zap [16], PDS [1] and Cells [2] etc. These projects successfully partition a single OS environment into multiple VMs more extensively and efficiently. However, very limited efforts have been made on application interactions cross-VM and intra-VM, although they are required by many cooperative applications. FVM [28] hard-codes a few IPC object names as exceptions in its virtualization layer, so that it can partially support the cross-VM communications between applications in a VM and in the host environment. However, it can not flexibly support the types of cross-VM communications that are not hard-coded. Moreover, it can not support other types of interactive operations, e.g., cross-VM names and intra-VM dependencies. As a result, it is able to only virtualize a limited number of ordinary Windows services, whereas, it can not virtualize Windows system services such as RPCSS and ordinary Windows services that require complex inter-service interactions, e.g. IIS service group. To virtualize system services, we propose a service virtualization scheme in our former work [22]. In contrast, this paper focuses on inter-application interactions. As service virtualization also involves inter-application interactions, some issues might be similar. However, the resolutions to these issues are different. Moreover, the most important issue addressed in this paper is on VM-VM interactions, while it was not considered in our former work.

Zap [16] introduces pods, which are groups of processes that are provided a consistent, virtualized view of the system. Processes outside a pod can only interact with processes inside the pod using network communication and shared files instead of IPC. Shuttle provides a possible solution to facilitate inter-application interactions across pod boundary when pursuing a better performance. Cells [2] is a virtualization architecture for enabling multiple virtual smartphones to run simultaneously on the same physical cellphone. It sets up IPC sockets to facilitate communication between VM and the host. Shuttle, however, also address issues related to VM-VM communication, cross-VM invocation and transferring hard-coded names. These extended functions may allow Cells to work in more application scenarios.

The second category of projects similar to our work focuses on how to achieve inter-VM communications for a hardware-level virtualization system. For example, the Xen [4] platform enables applications to transparently communicate across VM boundaries using standard TCP/IP sockets and traversing the network communication path via Dom0. In order to improve the performance of cross-VM communications, XenSocket [30], IVC [9], XWay [11], XenLoop [27] and Fido [6] have exploited the inter-domain shared memory provided by the Xen hypervisor. However, these techniques in Xen intercepts outgoing network packets beneath the network layer, and thus can not handle the inter-process communication (IPC) in OS-level VM as most IPC data will not go beneath the network layer. Different from these projects, Shuttle focuses on facilitating IPCs between OS-level VMs. Moreover, it handles not only cross-VM communications but also cross-VM names and startups, as well as intra-VM dependencies, which are mainly resulted from the characteristic of OS-level virtualization when multiple VMs share the single OS kernel.

The third category of projects is library operating systems [7][8][12]. The idea is that the entire *personality* of the OS on which an application depends runs in its address space as a library. A recent project, Drawbridge [19], shows the library OS can offer better system security and more rapid independent evolution of OS components. As a structuring principle, Drawbridge identifies three categories of services in OS implementations: *hardware services*, *user services*, and *application services*. Then, it uses these service categories to drive the refactoring of Windows into the Drawbridge library OS. Drawbridge packages application services into the *library OS* and leaves user and hard-ware services in the *host OS.* The scheme of Shuttle to handle inter-process interactions across VM boundary can be useful to resolve the multi-process applications problem in Drawbridge.

9 Conclusion

Advances in OS-level virtualization technology have strengthened the isolation between VMs. However, many interaction-intensive applications require penetrating the isolation boundaries to cooperate with the applications in other VMs. In this paper, we make the first step towards supporting the application interactions in an OS-level virtualization system by facilitating four types of interactive operations, including cross-VM inter-process

communications, cross-VM name transfers, cross-VM application invocations and intra-VM application dependencies. Specifically, we design a novel approach, Shuttle, that consists of four techniques, each of which intends to handle a corresponding interactive operations. As a result, a number of interaction-depending applications that can not run within a VM previously, e.g., RPCSS, IIS, can now run under the support of Shuttle. Empirical performance measurements on the prototype implementation of the proposed Shuttle approach show that the compromise on isolation are negligible and the additional performance overhead is rather minor, when compared with that of the original version of FVM.

Acknowledgement

We would like to thank our shepherd Galen Hunt and all the anonymous reviewers for their insightful comments and feedbacks. This work is supported by Natural Science Foundation of China under grants No. 60703103 and No. 60833005, US National Science Foundation under grants CNS-0751121, CNS-0751121 and CNS-0628093.

References

[1] B. Alpern, J. Auerbach, V. Bala, T. Frauenhofer, T. Mummert, and M. Pigott, “Pds: A virtual execution environment for software deployment,” in *Proceedings of the 1st International Conference on Virtual Execution Environments*, 2005.

[2] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (SOSP '11). ACM, New York, NY, USA, 173-187.

[3] M. Armbrust, A. Fox, R. Griffith et al., Above the Clouds: A Berkeley View of Cloud Computing, Unversity of California, Berkeley, Berkeley, CA, 2009.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177. ACM Press, 2003.

[5] A. Bavier , M. Bowman , B. Chun , D. Culler , S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, M. Wawrzoniak, Operating system support for planetary-scale network services, *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, p.19-19, March, 2004, California.

[6] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson, “Fido: Fast inter-virtual-machine communication for enterprise appliances,” in *Proceedings of the USENIX Annual Technical Conference*, San Diego, USA, 2009.

[7] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, 1994.

[8] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr.. 1995. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles* (SOSP '95), Michael B. Jones (Ed.). ACM, New York, NY, USA, 251-266.

[9] W. Huang, M. Koop, Q. Gao, and D.K. Panda. Virtual machine aware communication libraries for high performance computing. In *Proceedings of SuperComputing*, Reno, NV, Nov. 2007.

[10] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, 2000.

[11] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim. Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In *Proceedings of the fourth ACM International Conference on Virtual Execution Environments*, 2008.

[12] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, *14* (7), 1996.

[13] Linux VServer, http://linux-vserver.org/Documentation, 2010.

[14] Linux VServer, http://linux-vserver.org/Problematic_Programs, 2011

[15] Microsoft. Process Explorer. http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx

[16] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI02)*, pages 361–376, Boston, MA, Dec 2002.

[17] P. Padala, X. Zhu, Z.Wang, S. Singhal, and K. Shin. Performance Evaluation of Virtualization Technologies for Server Consolidation. *Technical Report HPL-2007-59*, HP Labs, April 2007.

[18] PC Magazine, PC Magazine benchmarks, http://www.pcmag.com/encyclopedia_term/0,2542,t=WebBench&i=48947,00.asp

[19] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. 2011. Rethinking the library OS from the top down. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (ASPLOS '11). ACM, New York, NY, USA, 291-304.

[20] D. Price and A. Tucker. Solaris Zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th Large Installation System Administration Conference* (LISA), USENIX, 2004.

[21] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, 63(9):1278-1308, September 1975.

[22] Z. Shan, T. Chiueh, and X. Wang. Virtualizing system and ordinary services in Windows-based OS-level virtual machines. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (SAC '11). ACM, New York, NY, USA, 579-583.

[23] S. Soltesz , H. Pötzl , M. E. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors, In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* 2007, March 21-23, 2007, Lisbon, Portugal.

[24] SWSoft, “OpenVZ - Server Virtualization,” 2006, http://www.openvz.org/.

[25] VMware. Vmware products. http://www.vmware.com/products/home.html.

[26] P. Walters, V. Chaudhary, M. Cha, S. Guercio Jr., S. Gallo, "A Comparison of Virtualization Technologies for HPC," In *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications* (aina 2008), pp.861-868.

[27] J. Wang, K.-L. Wright, and K. Gopalan. Xenloop: A transparent high performance inter-VM network loopback. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing* (HPDC), 2008.

[28] Y. Yu, F. Guo, S. Nanda, L. Lam, T. Chiueh, ``A Feather-weight Virtual Machine for Windows Applications", in *Proceedings of the 2nd ACM/USENIX Conference on Virtual Execution Environments* (VEE'06), June 2006.

[29] Y. Yu, H. K. Govindarajan, L. Lam, T. Chiueh "Applications of Feather-Weight Virtual Machine", In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (VEE08), Seattle WA, March 2008.

[30] X. Zhang, S. McIntosh, P. Rohatgi, and J.L. Griffin. Xensocket: A high-throughput interdomain transport for virtual machines. In *Proceedings of Middleware*, 2007.