Virtualizing Operating Systems for Seamless Distributed Environments[1]

Tom Boyd and Partha Dasgupta
Department of Computer Science and Engineering
Arizona State University
Tempe AZ, U.S.A.

Abstract

Applications and operating systems can be augmented with extra functionality by injecting additional middleware into the boundary layer between them, without tampering with their binaries. Using this scheme, we separate the physical resource bindings of the application and replace it with virtual bindings. This is called virtualization. *We are developing a* virtualizing Operating System (vOS*) residing on top of* Windows NT, *that injects all applications with the virtualizing software.*

The vOS makes it possible to build communities of systems that cooperate to run applications and share resources non-intrusively while retaining complete application binary compatibility. In this paper, we describe the structure, architecture and operation of the virtualizing Operating System supporting our virtualization concepts and methodologies

Keywords: Parallel/distributed computing systems, API Interception.

1. Introduction

The promise of global, seamless distributed systems, constructed out of many autonomous workstations has not materialized. This paper presents a design and preliminary implementation towards making such a system possible within a set of uniformly administered machines.

There are three major challenges hindering the development of distributed operating systems that bring seamless distribution to the desktop. The first challenge is the magnitude of change required for enhancing or adding to any of the system's capabilities.

The second challenge is the unavailability of applications for such distributed operating systems. If applications have to be modified and/or rewritten to take advantage of the distributed substrate and the distributed operating system, then the approach is doomed to fail.

The third challenge is the legacy nature of current systems and applications. Any changes to the operating system functionality leads to adding newer application programming interfaces (APIs). Few, if any, applications are rewritten to use the newer APIs.

The resolution to these challenges is through the *unobtrusive injection* of new functionality into existing systems. This approach requires no changes to the operating system or the existing application base, and yet endows the system with additional functionality that can be made as transparent or opaque to the end user as is necessary.

Using this approach, regular shrink wrapped applications can be run on regular standard operating systems, yet the underlying system can be a set of autonomous machines, providing a seamless distributed environment.

1.1 Computing Communities

Our research is part of a larger project called "*Computing Communities*" (or *CC*) [1]. The goal of the *CC* project is to enable a group of computers to behave like a large community of systems. The community grows or shrinks based on dynamic resource requirements through the scheduling and moving of processes, applications and resource allocations between systems—all transparently.

The computers participating in the *CC* utilize a standard operating system and run stock applications. The key technique to achieve such a system is the creation of a "*virtualizing Operating System*" or *vOS*. The main theme in the *vOS* is of course "virtualization", which is the decoupling of the application process from its physical environment. That is, a process runs on a virtual processor with connections to a virtual screen and virtual keyboard, using virtual files, virtual network connections, and other virtual resources. *The vOS has the ability to change the connections of the virtual resources to real resources at any point in time, without support from the application*. The *vOS* implements the functionality to

[1] This research is partially supported by grants from DARPA/Rome Labs (F30602-99-1-0517), Intel, and Microsoft, and is part of the "*Computing Communities*" project, a joint effort between Arizona State University and New York University.

The *vOS* implements the functionality to virtualize the resources by controlling the mapping between the physical resources (seen by the operating system) and virtual handles (seen by the application). The virtualization provided by the *vOS* can provide a plethora of advantages:

- The users can move their virtual "home machines" at will, even for applications that are currently executing.
- A critical service running on machine M_1 can be moved to machine M_2 if M_1 has to be cast away.
- Schedulers can control the complete set of resources.
- Applications can use resources, transparently aggregated from several machines. For example, a memory-intensive application can use memory in remote machines.

Our current work is based on the Windows 2000 operating system (but is extensible to any stock operating system). Windows 2000, like other operating systems, is structured such that the applications and the operating system contain a clearly delineated point of indirection, which is easily exploitable to add or interpose a layer of middleware.

The structure of the remainder of this report is as follows: Section 2 provides a general description of the virtualizing Operating System and the virtualization components. Section 3 describes the architecture of the system. The implementation of the system is described in section 4 with current status detailed in section 5. Section 6 describes related work and section 7 summarizes this work.

2. Virtualizing Operating System

The central mechanism that provides the features and benefits of this approach is the *virtualizing Operating System* or *vOS*. The main *vOS* theme is "virtualization", which is the decoupling of the application process from its physical environment. The core of the *vOS* operation is the *virtualizing System Manager* (*vSM*), which is a central management facility providing global coordination and control services (figure 1). To ensure that the *vOS* is scalable and to reduce issues of *vOS* fault tolerance, multiple *vSM*s may operate as peers and coordinate activities between their respective domains.

The *vSM* is located at one place, anywhere in the network and performs global functions. It works with the *virtualizing EXecutive* (*vEX*), a system command and control component residing on each participating system. The *vEX* is a Windows NT service acting as the *vSM*'s local agent and proxy.

The *vEX* uses and manages local workstation resources in combination with an API wrapper tool and the *virtualizing INterceptor* (*vIN*) to capture and administer the workstation processes that participate in the *CC*.

The *vBUS* performs the communication function between the system components. It is designed to provide support for different intercommunication requirements including message priorities, multicasting/broadcasting and point-to-point operations.

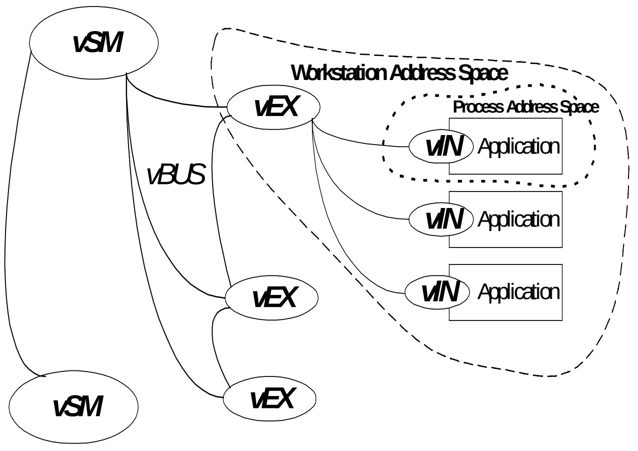

Figure 1: *vOS* system hierarchy

2.1 API/DLL Interception

As stated above, the *vIN* is responsible for capturing and then recording or reinterpreting much of the interactions between a running application and the underlying operating system. The power of virtualization comes from the reinterpretation of system calls and hence the capturing of system calls is a crucial underlying mechanism for our virtualization scheme. This capture is done by a scheme called *API Interception.*

API interception is gaining popularity in systems programming as it has unlimited potential for augmenting system functionality in a non-intrusive fashion. Most operating systems allow API interception methods to be built at the user level. In the Windows 2000 DLL scheme, when the application is loaded, the API references are resolved to a table of addresses in the user space called the *Import Address Table* (IAT), and filled in at run time [2]. The DLL contains a list of exported addresses used to populate the table. Using an indirect pointer, the application jumps to the API entry point within the DLL. By modifying the addresses contained in the IAT, the application call is redirected to an alternate API entry point.

2.2 Handle Virtualization

The Windows 2000 system is architected to use handles as references to most every component and resource of the system. The files, network and communications, processes, threads, fibers, events, windows, menus, submenus, edit buffers are just a few of the resources that have handles associated with them.

To virtualize applications and resources requires creating and mapping new handles and replacing references within API calls between systems (figure 2). Virtual handles allow each API to function correctly on the local system as well as forming the basis for abstracting resource from specific system instances.

Handles normally consist of a 32-bit value. To aid in tracking and debugging, the handle is encoded with an origination code. The code includes an identifier for the source machine, process, thread and handle type. This information is useful for tracing or debugging a migrated process especially after several iterations.

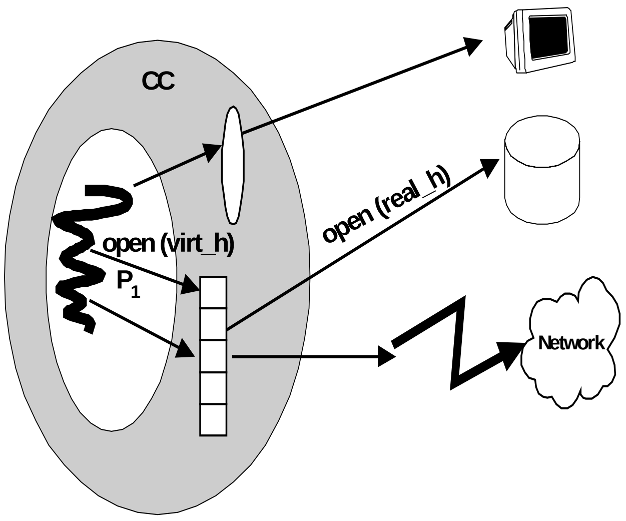

Figure 2: Handle virtualization

3. Architectural Overview

The main architectural components of the *vOS* are the *vSM*, *vEX*, *vIN*, and *vBUS*, as introduced in section 2. The following sections provide a review of the main architectural features of each of these components (figure 3).

3.1 Virtualizing System Manager

The central component of the *vOS* is the *vSM*. It is a control process residing on any system within the *vOS* domain. The *vSM* is the primary interface with the user for system status reporting, command and control, system initialization and shutdown. The *vSM* is a window that reports current state and activities of the *vEX*s, *vIN*s and system resources. The *vSM* carries the role of central and primary controlling agent and information source for the system. Each *vEX* communicates with the *vSM* to acquire knowledge of other *vEX*s and system resources. The *vSM* is located using the Windows 2000 Active Directory DNS service.

3.2 Virtualizing Executive

Each system platform participating in the *vOS* contains a system executive (*vEX*) process. The *vEX* provides system level coordination and control. It acts as the common communication point for each of the *vIN* instances within the scope of one physical platform. A view of local and remote resource, local system activities, security and policy are maintained and managed by the *vEX*.

vEX is a multithreaded NT service. Multiple threads handle the *vBUS*, command and control, local user interface, resource, migration, policy and failure management functions. The *vEX* is autonomous and quasi-persistent. If a local system failure occurs, the *vEX* can checkpoint its own, the *vIN*'s and the application's states and can migrate or be migrated elsewhere within the *vOS*. It performs this role by exchanging information periodically with the *vIN*s and the *vSM*.

3.3 Virtualizing Interceptor

The *vIN* is the interception middleware component. One *vIN* is required for each process or application participating in the *vOS*. *vIN* captures process information through the API interception mechanism. Several threads are established within the process state to handle: the *vBUS* IO, command and control, watchdog and trace and API interception. Each *vIN* communicates with the local system executive, *vEX*, for command and control directives. Virtualization tables[2] are built and maintained by the *vIN*. API calls creating, using or returning handles will always use a virtual handle created and maintained at this level. Where necessary it performs message marshalling, unmarshalling, forwarding and reception. Since process migration requires lower layer recreation or reassignment of handles, the virtual handle references are sent as part of the application's state information.

3.4 Virtualizing BUS

The *vOS* is a single logical system constructed from multiple individual workstation components. Its overall performance depends upon effective interaction between the workstation instances. Efficient and timely sharing and exchange of information, status and messages flows between and throughout the *vOS* environment is important and is definable in terms of singular, group and universal relationships. The virtualizing BUS (*vBUS*) provides the support for the overall system operation within the context of the Windows NT and general Internet environment.

The virtualizing BUS is architecturally similar to a hardware bus in terms of the approach to the logical presentation and control of data. It uses a simple and efficient API for delivery and reception of *signals* and *messages*.

[2] *vOS* tables that contain the virtual to real handle relationships and other information such as handle or resource type and handle specific data such as addresses.

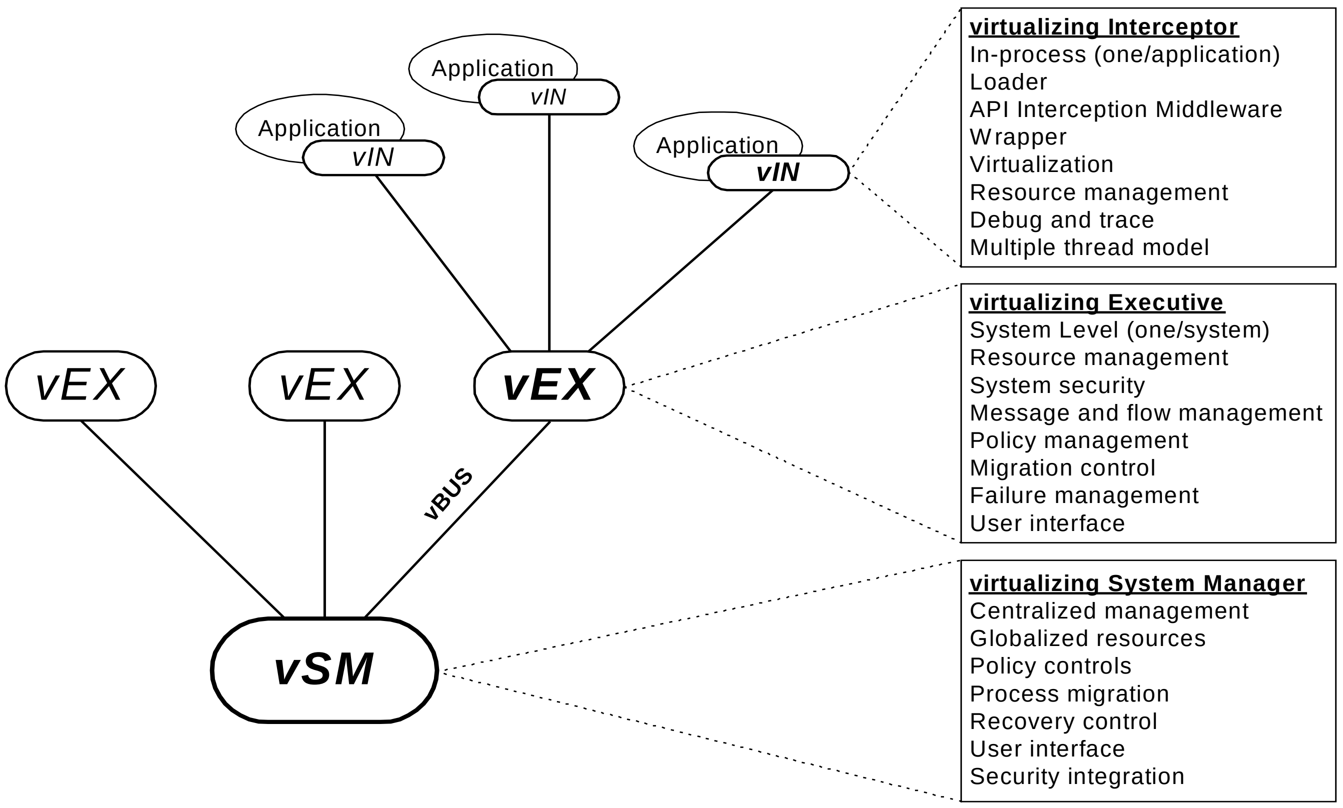

Figure 3: vOS Architecture

The architecture of the *vBUS* utilizes existing transport facilities: IP, TCP, UDP and other inherently available capabilities. With the introduction of RFC based multicast and broadcast protocol support, new opportunities are available to use facilities such Multicast Backbone technology [3].

4. *vOS* System Archetype

We are building the virtualizing system using components and lessons learned from our prior work. The development proceeded from the initial development of the *vIN* as the foundation for the system. We next constructed the basic *vEX* using a simplified method to develop the *vEX*/*vIN* relationship.

4.1 *vSM* Implementation

Upon initialization the *vSM*, presents the user interface window, initializes the *vBUS* and sets its service availability in the DNS. It waits until contact is made on a listening socket then accepts and begins communication with the contactor, which is a *vEX*. The *vEX* reports its status and resource information which is recorded in tables organized by *vEX* and *vIN* combinations. Resources are assigned currently using a persistent table, which can be altered through the user interface

The user interface consists of a set of preset command selections and display areas. *vEX*, *vIN* and *vBUS* information are available for display based on the selection made. Commands and control entries are selected from a static list. Additional information is requested depending upon the command. An automated periodic update request and display is available for a pseudo real-time update of the system status.

Currently, the *vSM* is manually launched. Since it will itself be migratable, we intend to build a more robust "homing service" that is capable of invoking and migrating the application without imbedding or tying the *vSM* to a single machine.

4.2 *vEX* Implementation

The *vEX* is started whenever the base operating system becomes active. After activation, it initializes the *vBUS* and locates the *vSM* service through a DNS request. The *vSM* is contacted on a well-known port and the *vEX* sends basic state, location and resource information to the *vSM* then receives state, control and policy information.

The *vEX* next establishes the well-known file, event and mutex names for usage by the *vINs*. Once these names are established, the *vEX* waits for signals from the *vINs* and the *vSM*. Status is reported by the *vEX* upon request from the *vSM*. Commands from the *vSM* are executed and the results reported back to the *vSM*.

Impersonation[3] is used by the *vEX* and *vIN* when processes or resources are migrated. This allows the security for the currently logged user to be passed between the systems without requiring open use of logon or password information.

When a process or resource migration is requested, the receiving *vEX* starts a stub process which, the *vIN* recognizes and sets up to load and build the requested application environment. After establishing communication with the local *vEX*, the *vIN* receives the remote *vIN* address, contacts the remote *vIN* and proceeds to perform the requested migration task.

4.3 *vIN* Implementation

To capture the initialization state information for an application, *vIN* loads and executes at some point prior to the initial entry into the application. This is called process interdiction. and is implemented by intercepting the *CreateProcess()* and *CreateThread* API calls in *kernel32.dll*, setting the SUSPEND flag then injecting the *vIN* into the application space. After the interception environment is established, the process is resumed and the state information is collected.

Communication is established by the *vIN* to the *vEX* by opening a memory mapped file using a well known file name, signaling an event using a well known event name then waiting on a well known mutex access to the memory area. The *vEX* is waiting for this event signal. The common file area is used to communicate control values and some data between the processes. Once the well-known event is received by the *vEX*, a new unique memory mapped file name and event signal value is placed in the common area, and the well-known mutex is signaled. The *vIN* uses the file name and event signal to establish normal communications with the *vEX*.

As the application begins execution, the API initialization and continuous state values are captured, virtualized or stored. The virtualization values or references are placed into a structured table for later usage.

4.4 *vBUS* Implementation

To perform network I/O, the *vBUS* uses a library of Winsock 2 functions. Threads are used to support callbacks, signals, command and control, point-to-point transfers and broadcasts.

Callbacks are implemented as a function list, event code, class and type. When *vBUS* determines an event has occurred, it matches the event code and class with the function and performs the callback type. Signals are implemented using an assigned port number and a *select* function call. A *send/receive* pair passing *void* is used to create the signal event. Callback with *void* is used on the receive side to complete the signal.

[3] Impersonation is a mechanism in Windows 2000 to allow a process to run under another user's id (similar to "su" in Unix.)

Command and control, which is set as the highest priority activity thread in the *vBUS*, uses the callback facility to pass command notification to the *vBUS* instantiator. It also recognizes a limited set of commands for internal control.

Point-to-point transfers are buffer and forward operations. Data sent by the instantiator is sent to a receiver and data received is provided to the instantiator. Broadcasts are currently implemented using the *FCAST* code from Microsoft Research. Currently a `Shutdown` command is implemented which successfully causes the *vEX*s to become inactive and the *vIN*s to terminate.

5. Status

The status of the implementation is a set of prototypes that show the feasibility of the approach. Each of the above-mentioned components (*vSM*, *vEX*, *vIN* and *vBUS*) exists as separate programs with limited facilities and they have been tested to work together in several situations, described below:

A system that tests the overall concept of the *vOS* is our "window cloning" testbed [4]. This testbed uses a stock application called *RegMPad* (available from the MSDN Library), which is a multiple document interface variation of *Notepad*. The system is capable of intercepting and migrating parts of *RegMPad* such that the window and mouse controls are moved to a target machine and the logic execution happens on both the target machine and the source machine (window message processing and menu processing on the target machine and the rest remain on source machine).

We have also extensively experimented with process migration of single threaded processes. In [5] we show how to migrate a process that has active network connections, using our approach. In [6] we show how to migrate processes, which are actively interacting with users on the screen (using *WinMine*). Similar tests have been done with processes using files.

We are currently working on incorporating all the pieces of software that has been built into a coherent system with a clear delineation between the various components and the ability to interoperate and merge the features. We have targeted a multithreaded, network *telnet* application for migration. We are currently unraveling the multithreaded nature of Win32 storage assignment and usage.

6. Related Work

There are a few production and a number of research systems available today that use API/DLL redirection for interpositioning middleware. API interception can be done using toolkits such as *Detours* [7] or *Mediating Connectors* [8]. Systems using such facilities include:

COP: COP uses Detours and is a collaboration between Microsoft Research and the University of Rochester [9].

It is MFC oriented building and wrapping components around the Win32 API and using a COM interface for intersystem communication.

NT-SwiFT: *NT-SwiFT* [10] also known in its first release from Lucent Technologies as *SwiFT for Windows NT*, provides six functional components: automatic error detection and recovery, check pointing/message-logging, fault tolerance, event logging and replay, data replications and IP packet re-routing. It is capable of migrating applications between systems and restarting them in the event of a failure. It is assumed the server application can and should be modified to incorporate the SwiFT capabilities and that client applications do not necessarily need to be modified.

Transparent Checkpoint Facility On NT: This system performs API interception by rewriting the IAT to forward calls to the check pointing software, which is implemented as a DLL [11]. It captures system and application state while the application executes then replays the captured state data when restarting an application. Automatic check pointing is installed on the application by using an alternate loader and does not require application change, however, APIs are available for an application to explicitly control the system behavior. The system has several acknowledged limitations, including the requirement that temporary file state be retained for a restart, applications that bypass the IAT may not work correctly and multiple interacting processes are considered too complex to handle.

7. Summary

In this paper, we have presented an operating system in support of our virtualization work. It describes a system that uses the same fundamental unobtrusion philosophy as the virtualization techniques employed. The *vOS* is hierarchically structured to allow for system extensibility, but it still provides locally autonomy for robustness.

This system provides the basis for additional research into the viability and functionality of a Computing Community. Further work is in progress to incorporate security and session persistence as well as application adaptation, which includes fault detection and fault tolerance.

8. References

[1] Partha Dasgupta, Vijay Karamcheti and Zvi Kedem**,** *Transparent Distribution Middleware for General Purpose Computations,* International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), June 1999.

[2] Jeffrey Richter, *Applications for Windows, Fourth Edition*, Microsoft Press, 1997.

[3] Stephen Deering, *MBONE-The Multicast Backbone*, CERnet Seminar, 3 March 1993.

[4] Tom Boyd and Partha Dasgupta, *Injecting Distributed Capabilities into Legacy Applications Through Cloning and Virtualization,* to appear in the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, June 2000.

[5] Ravikanth Nasika and Partha Dasgupta, *Transparent Migration of Distributed Communicating Processes,* to appear in the 13th International Conference on Parallel and Distributed Computing Systems, August 2000.

[6] Raghavendra Hebbalalu, *File Input/Output and Graphical Input/Output Handling for Nomadic Process on Windows NT*, Master's Thesis, Arizona State University, August 1999.

[7] Galen Hunt and Doug Brubacher, *Detours: Binary Interception of Win32 Functions*, Proceedings of the 3rd USENIX Windows NT Symposium, July 1999.

[8] Robert Balzer, *Mediating Connectors*, Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop, 1994, ISBN 0-262-57104-8.

[9] Robert J. Stets, Galen C. Hunt, Michael L. Scott, *Component-based APIs: A Versioning and Distributed Resource Solution*, IEEE Computer, 32(7), July 1999.

[10] Yennun Huang, P. Emerald Chung, Chandra Kintala, Chung-Yih Wang De-Ron Liang, and De-Ron Liang, *NT-SwiFT: Software Implemented Fault Tolerance for Windows NT*, 2nd USENIX Windows NT Symposium, July 1998.

[11] Johny Srouji, Paul Schuster, Maury Bach, Yulik Kudmin, *A Transparent Checkpoint Facility on NT*, Proceedings of the 2nd USENIX Windows NT Symposium, August 1998.

<u>*Sponsor Acknowledgment:*</u> Effort sponsored by the Defense Advanced Research Projects Agency and AFRL/Rome, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0517. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.

<u>*Sponsor Disclaimer:*</u> The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, AFRL/Rome, or the U.S. Government.