Injecting Distributed Capabilities into Legacy Applications Through Cloning and Virtualization[1]

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

Partha Dasgupta
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 completely non-intrusively while retaining complete application binary compatibility.

In this paper, we describe a prototype system that virtualizes the application's window, making it possible to relocate the window to remote machines without the application's awareness. The prototype copies, or clones *a window of an application onto a display on a remote machine and then, using* API interception, *applies the application semantics to the clone window in terms of data and message flow.*

The virtualization of the application's window is one of the steps towards making all system resources virtualizable and any application movable between systems. This research is part of a larger project called Computing Communities *(*CC*) which is building large unions of distributed machines supporting shared resource management using legacy applications.*

Keywords: Distributed Operating Systems, Middleware, Process Migration, API Interception, Windows NT.

1. Introduction

Transparent support for distributed and mobile applications in current computing architectures is made difficult due to two major infrastructural challenges. The first challenge is the magnitude of change required for enhancing or adding to any of the system's capabilities. If the operating system or runtime system is altered, the potential exists to create a system wide cascade of application modifications, which becomes unavoidably expensive. Decision-makers are unlikely to embark on massive system changes if there is little return on the investment or if it is considered excessively intrusive.

The second challenge to system augmentation is the legacy nature of current systems and applications. As complexity has increased, the ability to make fundamental changes to the system has dramatically decreased. Changes and additional functionality leads to adding newer APIs (application programming interfaces). In recent years, we have seen the proliferation of new, so-called standard APIs, creating literally thousands of APIs and variants and hampering innovation. In addition, few, if any, applications are rewritten to use the newer APIs, further impeding progress.

The solution to both 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.

In this paper, we show how such injectable functionality is achievable and how it is used to create services that allow programs to become mobile in a distributed environment. In addition, we discuss our unobtrusive injection approach as applicable to the Windows NT operating system and its applications. We provide details of a prototype software package we have implemented to

show feasibility through the techniques called *window cloning* and *API interception*.

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 act like a large community of systems, which 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 novelty of the *CC* approach is that it is non-intrusive, causing no application redesign, re-coding or recompiling. Binary compatibility is assured while adding new services and features such as *transparent distribution, global scheduling, fault tolerance,* and *application adaptation.*

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 virtualize the resources by controlling the mapping between the physical resources (seen by the operating system) and virtual handles (seen by the application). In general, virtual handles represent the software resources like file handles, graphics handles and network handles (Figure 2). The application uses the virtual handles as if they are OS generated. When the application passes a virtual handle to a system call, the *vOS* intercepts that call and passes the actual physical handle to the system call. This enables the applications to use remote resources as though they are local and change the mapping between the virtual handles and physical handles dynamically. In essence, the *vOS* provides a unified virtual machine interface for the applications. This environment consists of virtual CPU, virtual communication subsystem, virtual user interface and virtual file system. The *vOS* depends on a number of mechanisms to provide the required services. The mechanisms used by the *vOS* include API intercepting, mapping of virtual handles to physical handles, GDI/file and network virtualization and process migration.

Hence, it is possible to redirect the output of the application from one display to another, or to move the application from one machine to another—without moving its screen or keyboard location. This leads to a plethora of opportunities as well as an innovative systems management.

1.2 Window Cloning and Virtualization

This paper concentrates on the virtualization of the application's window to allow the movement of its presentation location at runtime. This is one of the key steps necessary in realizing the *CC* system.

The window is a Graphical User Interface normally bound to a specific instance of a system and application. In our implementation of window virtualization, we focus on creating a *clone* of an existing window then show that the clone is an active virtualized component by connecting the application semantics to the cloned window. An alternative method to window cloning uses a form of "bit-scraping" (see section 3.1). We, however, use the virtualization approach, which is more flexible and applicable uniformly to all resources, not just windows.

The rest of the paper is organized as follows. Section 2 discusses the API interception technique, section 3 describes our approach to window virtualization, section 4 describes our prototype, section 5 describes the lessons learned and section 6 provides details of related work.

2. API Interception Mechanism

The Windows NT system provides a late binding API architecture through *Dynamic Link Libraries* (DLLs) [3]. Late binding creates the opportunity to insert a middleware [4] component into the system that can examine, modulate or replace the API call [5]. The insertion of the middleware component creates the API interception mechanism.

API interception forms the basis for the injection of system functionality in *CC*. By inserting

code between the application's API call and the system API, new functionality is introduced.

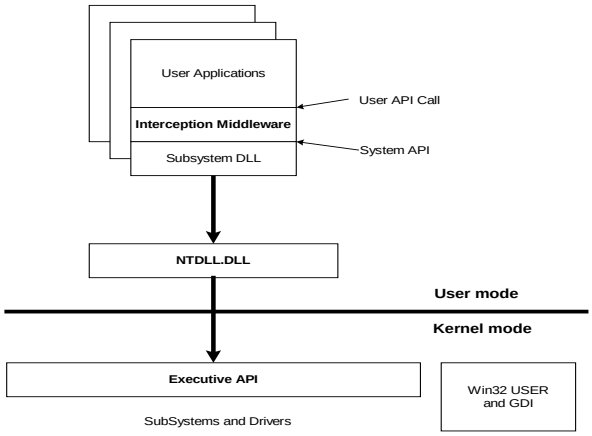

Figure 1: API Interception

In the Windows NT 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. 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.

The new API code acts as a *wrapper* around the existing application code and has access to all of the data explicitly passed and returned by the API call. New functionality is added through the manipulation of the data in these calls and state information is captured for subsequent usage.

The ISI *Mediating Connectors* toolkit automates the process of installing the API interception applications [6]. To create the interception code or wrapper, a DLL is developed containing the new API calls. A text file is created that defines the Wrapper to API mapping for the wrapper load routine. API calls are provided to explicitly load and install, uninstall and unload the wrappers. Any system or user DLL is wrappable using this approach.

2.1 Handle Virtualization

The Windows NT 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. These handles are unique to each system and as such are not system interchangeable.

To virtualize applications and resources requires creating and mapping new handles and replacing references within API calls between systems. 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. Although handles are extractable during the system operation, they are best captured and virtualized as they are created by system calls.

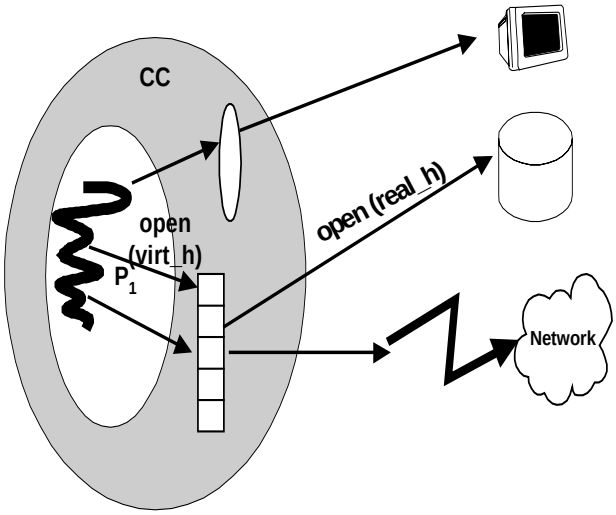

Figure 2: Handle virtualization

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.

3. Approaches to Window Cloning

Our objective is to clone the window of an application onto a display located at a remote computer, and to achieve this using virtualization. This approach enables us to define and develop the techniques required to virtualize additional resources as we work toward creating a *vOS* supporting the requirements of the *CC* design.

The Windows NT system uses a graphics engine to manage the mouse, keyboard and display. The engine is notified at window creation what components exist in the window, where they reside and what to tell the application when an action such as a mouse movement or menu selection occurs. The application is given a first chance opportunity to take an action on the event. In this way the application interacts with the Windows engine and through the use of a set of APIs, the window and the data contained therein are manipulated.

3.1 Cloning By Means of Bit Scraping

There is a technique employed by commercial applications such as NetMeeting to project a Window, from one physical system to another through the use of a "*bit-scraping*" technique. Images are formed as a set of bits in memory for display on the monitor. The device drivers and adaptors interpret the bits to build a final composite video output image. Actions such as mouse movements and menu selections cause bits to be changed at the appropriate locations in the map so that the next display of the bit map will appear to show an image movement of the mouse or a menu dropped down. By periodically copying or "scraping" the bit map memory and sending some or all of the bits to a proxy application on a receiving machine, the bits are windowed and displayed as if they are a resource local to the remote machine.

In the bit-scraping scheme all the windows operational semantics, even the low-level primitives, remains within the original machine's applications and the remote machine simply displays the bits. This approach is inefficient, requires significant network traffic and makes the program on the original machine immovable.

Since we are interested in the essence of "freeing" the program from the physical machine, we perform the same function by virtualizing the window interface.

3.2 Cloning via Virtualization

Through cloning of the window, we are able to move the window's semantics with the clone. Thus, much of the logic of displaying and updating the window is performed at the remote site and is decoupled from the application's original host. A window's state, location, dimension, content and low level context are readily accessible through the NT API calls.

A running application is injected with a "cloning procedure" and the application's window information is collected and sent to a proxy executing on a remote system. The proxy on the remote system uses the collected data and reconstructs an exact visual duplicate or clone of the original window.

By connecting the cloning procedure to the proxy, the message traffic generated by the local mouse and keyboard flows between the proxy and the original application. Since the proxy has full control over the clone window (remote), Window engine facilities such as the Multiple

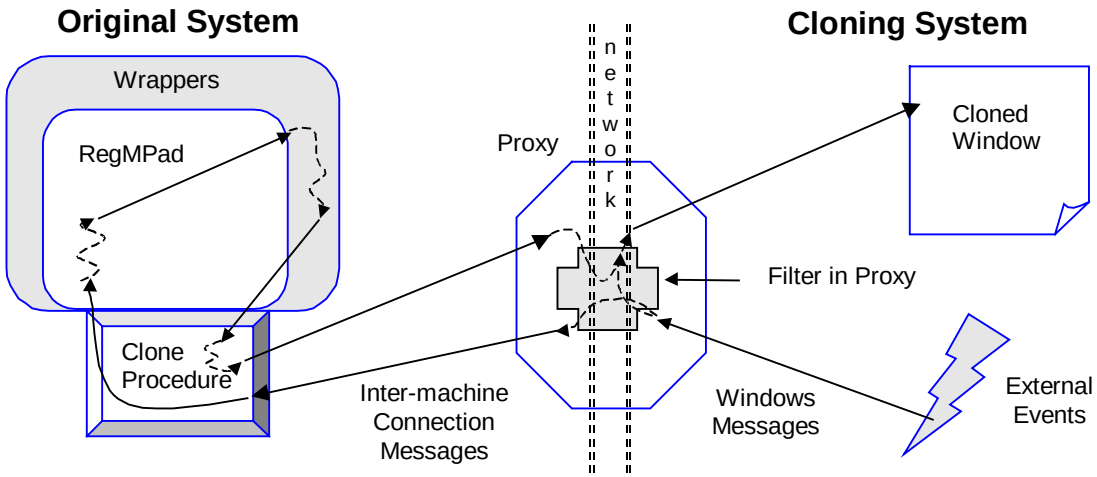

Figure 3: The interactions between the clone procedure, proxy and Windows

Document Interface (MDI) can be used at the remote site to perform standard control and editing tasks.

Actions such as menu selection and file selection requests are intercepted by the cloning procedure and sent to the original application for interpretation and action. API calls generated by the original application are intercepted using wrappers and the calls, where appropriate, are redirected to the proxy for API completion.

4. Prototyping with *RegMPad*

We have built prototype software that clones windows from a standard application. The testing application used was *RegMPad*, an example text editor available from the MSDN Library. *RegMPad* edits text using the MDI (multiple document interface) and hence has a sufficiently complex variety of window behaviors and structures for our study. Access to *RegMPad* source helped us to implement the cloning software—no changes were made to *RegMPad*.

As described in Section 3.2, a running *RegMPad* session is frozen by dynamically injecting a cloning procedure into it. The cloning procedure starts a new thread in the *RegMPad* application. This new thread establishes communication with a proxy process and sends it profile data for each of the windows and sub-windows existing within the *RegMPad* session. The thread then loads the API wrappers to handle the API actions of the *RegMPad* process on the original system.

At this stage, there are four entities to orchestrate. The Windows engine on the cloning system sends a stream of events to the proxy in response to user actions. The proxy filters these events, sends some of the messages to the *RegMPad* process via the cloning procedure, and handles some of the messages locally on the cloning system. The cloning procedure sends some of the windows messages to the *RegMPad* application and the wrapper surrounding the *RegMPad* application catches the results and sends them to the proxy for display.

Determining which messages and calls to process and where they should be processed required an extensive study of the messaging interface, the MDI routines and the presentation techniques used by Windows. This was accomplished by observing both the message traffic from the Windows engine as well as the associated API calls generated by the messages.

4.1 Prototype Components

The prototype system consists of 5 major developed components plus the *RegMPad* application. The following provides a general description of each of these developed components:

- *InjectLibrary*: Contains the mechanism for overall system operation and control through the user command and control interface window. It injects *RegMPad* with code, across the process boundary, which loads the *Migrate* code into the application space. It receives window profile information from the *Migrate* process, builds the window, acts as the default class procedure for the newly instantiated clone window, creates the window procedural threads for the window commands and the forwarded API requests.
- *Migrate*: Contains the DLL code loaded by the process injected from *InjectLibrary*. Once the DLL executes, it establishes a communication connection with the *InjectLibrary* then performs the actions requested including wrapping the application, capturing the clone window information, sending the information to *InjectLibrary*, migrating the window semantics and API interceptions by way of two threads to *InjectLibrary* then destroying the window when finished.
- *ProcessConnector*: Contains the ISI wrapper logic in the form of a DLL. It is loaded in the *RegMPad* process space when *Migrate* makes the wrap process request. It contains the function logic for 44 API calls identified specifically for the *RegMPad* application. These calls work in conjunction with a shared set of marshalling and unmarshalling routines to exchange the API calls data with the *InjectLibrary* call handlers.
- *BUSsw:* Is a shared set of routines based on named pipes that provides the basis for communication between the system components. It operates as a separate thread within the *InjectLibrary* and *Migrate* process spaces. *BUSsw* is structured to operate as a companion set of client/server applications.
- *InjectLibrary Proxy:* A specialized portion of the *InjectLibrary* application that performs a

proxy operation for the full *InjectLibrary* when operating across two systems. In normal operational mode, *InjectLibrary* expects the process to be on the same machine. The proxy process acts as a remotely controlled version of the *InjectLibrary* routine.

5. Status and Experiences

The *RegMPad* application works well with our procedures, proxy and wrappers. We are now extending the software to handle more events to enable it to work with any application (*RegMPad* utilizes a subset of the Windows facilities). In the process of building this prototype, we have learned of several issues important to the building of a general-purpose window virtualization tool. These issues are described in the following subsections.

5.1 Catch API Traffic Early

Our initial approach to collecting the target application's window state was to inject the target with the state collector after the application had been operational for some length of time. This approach has difficulty identifying certain types of minor information because it is not readily available through the Windows API calls. Information such as the placement of an element in a menu is best captured as it is being created—hence the application should be wrapped prior to running.

5.2 API Content

Blindly capturing and forwarding API calls is insufficient. For example, Windows messages are sent using a single API call, *SendMessage*. This single API interface contains a multitude of individual requests and references that each needs to be examined and marshaled. Decisions are made by the cloning system whether to handle the request locally, remotely or not at all based on policy, state, as well as the new functionality requirements. For example, the proxy should handle mouse movements and window presentation. Mouse menu selections most likely are destined for the actual application. Grey areas include issues such as where to perform file I/O and are candidates for policy decisions. It may be beneficial to duplicate certain requests on both systems.

If file, application, window, network and keyboard/mouse resources are virtualized, then the careful choice of locality of reference and functionality becomes very important.

5.3 API Reentrancy

NT Windows is not completely reentrant with respect to window message handling within the engine. For example, a cloned window processes a request that is subsequently forwarded back to the original program. The original program generates an API backflow request to the clone window process. Since a request is already pending in the clone window it can potentially block the backflow API request. Careful request and release procedures need to be implemented to prevent this type of backflow deadlock.

5.4 Multi-Interface Data Flow

There are at least three separate inter process data and message flow types requiring different types of specialized support between systems:

- Command and control
- Proxy or "flow through"
- API message traffic

Command and control are directed, priority messages used by the controlling application to manage the various process components such as the injector and proxy setup and state changes. Proxy messages represent a multi-sourced form of data pipelining with an ordering requirement between the injected and cloning systems but with no data interpretation. API message traffic is also ordered traffic that requires marshalling and unmarshalling with special handling or interpreting. Each of the message types normally requires a response.

6. Related Work

There are a number of production and a few research systems available today that provide for the projection of windows over a network to different machines.

6.1 Commercial Applications

NetMeeting from Microsoft provides application projection between multiple PCs over a network. The window is captured at the GDI level using a special standards based driver. *pcAnywhere*, from Symantec Corporation, *Carbon Copy 32* from Compaq and *CoSession Remote 32* from Artisoft Corp provides access and control of a remote PC. The window image is captured at the GDI layer

and projected using a custom driver. Citrix System's *MetaFrame* provides application server software for the Microsoft NT Server, Terminal Server Edition. It supports multiple applications executing on a single server with window projection to a variety of thin clients using a standards based driver. All the above systems use some form of bit scraping employing device drivers embedded in the kernel. This is different from our approach, in both design and implementation—we do not use kernel drivers.

6.2 *Detours*

Detours [7] from Microsoft Research is an API interception application library that uses DLL delayed binding in conjunction with API preamble rewriting to intercept and redirect application calls. The redirected Win32 function call is routed through the detour code where it still has access to the original function through a *trampoline* function. Rewriting application code creates several potential security, portability and compatibility issues, so this approach is not used in the *vOS* implementation.

6.3 *COP*

COP [8], a collaboration between Microsoft Research and the University of Rochester, has a similar design goal as our research. It is MFC oriented, building and wrapping components around the Win32 API and using a COM interface for intersystem communication. It uses *Detours* for API interception.

7. Conclusions

Through the understanding of the elements, issues and techniques involved with virtualizing a Windows NT Graphical User Interface, the initial steps towards creating a virtualizing Operating System have been taken. We have determined that it is possible to inject functionality between the application and the operating system unobtrusively, thus opening the door to the addition of new behavior and capabilities.

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] Michael B. Jones, *Interposition Agents: Transparently Interposing User Code at the System Interface*, Proceedings of the 14th Symposium on Operating Systems Principles, pp. 80-93, ACM Press, December 1993.

[3] David A. Solomon, *Inside Windows NT, Second Edition*, Microsoft Press, 1998.

[4] Philip A. Bernstein, *Middleware: A Model for Distributed System Services*, Communications of the ACM, 39(2), February 1996.

[5] Eric M. Doshofy, Nenad Medvidovic, Richard N. Taylor, *Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures*, Proceedings of the 1999 International Conference on Software Engineering, May 1999.

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

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

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

[9] Jeffrey Richter, *Advanced Windows, Third Edition*, Microsoft Press, 1997.

[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.

Sponsor Acknowledgment: 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.

Sponsor Disclaimer: 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.