# Process Migration: A Generalized Approach Using a Virtualizing Operating System[1]

Tom Boyd and Partha Dasgupta
Department of Computer Science and Engineering
Arizona State University
Tempe AZ, U.S.A.
{tboyd, partha}@asu.edu

## Abstract

*Process migration has been used to perform specialized tasks, such as load sharing and checkpoint/restarting long running applications. Implementation typically consists of modifications to existing applications and the creation of specialized support systems, which limit the applicability of the methodology. Off the shelf applications have not benefited from process migration technologies, mainly due to the lack of an effective generalized methodology and facility. The benefits of process migration include mobility, checkpointing, relocation, scheduling and on the fly maintenance. This paper shows how regular, shrink-wrapped applications can be migrated.*

*The approach to migration is to virtualize the application by injecting functionality into running applications and operating systems. Using this scheme, we separate the physical resource bindings of the application and replace it with virtual bindings. This technique is referred to as* virtualization. *We have developed a* virtualizing Operating System (vOS)*, residing on top of Windows 2000 that injects stock applications with the virtualizing software. It coordinates activities across multiple platforms providing new functionality to the existing applications. The vOS makes it possible to build communities of systems that cooperate to run applications and share resources non-intrusively while retaining application binary compatibility.*

## 1. Introduction

Conventional approaches to process migration depend on specially written migratable applications, which utilize distinct migration libraries and sometimes require a restricted set of operating system functions. Other approaches require modifications to the operating system kernel in order to facilitate migration. Such migrations are targeted to applications that are involved with parallel computations or long running numeric algorithms.

General purpose off the shelf (binary only) applications cannot benefit from these specialized migration facilities. Traditionally, it has not been considered necessary to endow applications, such as word processors, spreadsheets, browsers with migration capabilities. However, having the ability to migrate provides immense advantages for all applications, especially in mobile and distributed environments (see section 1.1).

In our research of virtualizing operating systems, we have established the need to be able to migrate any running application, without access to its source code. This paper describes the process which we adopted to build migration facilities, by decoupling the application from its environment (virtualization). Our approach allows legacy applications to participate in newer, "seamlessly distributed" computing environments, transcending process migration.

The virtualization mechanism depends upon API interception methodology. The APIs of an application are typically serviced by library routines. Library routines are connected to the application using a layer of indirection. This layer of indirection presents an opportunity to capture and modulate the API call through the modification of the API call parameters. These modifications include the ability to capture, modify and restore the state information, inherently available within the API call, through the implementation of a system which captures and restores state information.

This research discusses both the implementation and the core work that enables existing applications to assume new and novel characteristics and behaviors. Specifically; we examine process migration using the virtualiz-

[1] This research is partially supported by grants from NSF (CCR-9988204), 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.

ing Operating System, developed to capture and manipulate the application API state data.

### 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 movement 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* [2,3]. The main theme in the *vOS* is "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 a 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). The virtualization and process migration 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 a set of CPUs in a preemptive manner.
- Applications can use resources, transparently aggregated from several machines. For example, processor-intensive applications can use the CPUs in remote machines.

### 1.2 Useful Innovations

Utilizing the *vOS* as the basis for further study, we are examining some of the more important aspects of a Distributed Operating System, including transparent distribution of resources, fault tolerance, application adaptation, synchronization, global scheduling and event synchronization; all of which are accommodated within the framework of the *vOS* architecture.

Process migration, one of the fundamental technologies required for moving applications between systems is described in this paper. Our research has shown that the processes and methods used to migrate applications can be structured into three fundamental approaches. One of these approaches is fully integrated into the *vOS* and through our work, we prove that the remaining work is feasible.

Although the current work is based on the Windows 2000 (Win2K) operating system, it can be applied any other stock system. Win2K, 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 paper is as follows: Section 2 provides the general description and an architectural overview of the virtualizing Operating System. Section 3 describes the process migration methodologies developed in this research. Section 4 focuses on the observed performance of the migration using the *vOS*, while section 5 describes related work. Section 6 summarizes this research.

## 2. Virtualizing Operating System

The *virtualizing Operating System* (*vOS)* is the central mechanism of our research. The key *vOS* technology is "virtualization," which enables the decoupling of the application process from its physical environment.

### 2.1 Architecture

The *vOS* is a hierarchically structured and distributed application-management system that provides key global coordination and control services for the *CC* environment. Each *vOS* is responsible for a group of machines reachable through the network which becomes its domain of control.

The *vOS* unobtrusively integrates its components into the existing Win2K system. It provides services that intercept and virtualize the existing applications on the member machines, without altering the application's code in any way.

The *vOS* system consists of three main components: the *virtalizing System Manage (vSM)*, the *virtualizing EXecutive (vEX)* and the *virtualizing INterceptor (vIN)*. Each of these components is placed at a key control point and provides the overall functionality of the *vOS*. The control points are located either at the machine or application level.

### 2.2 Components

The *virtualizing System Manager* (*vSM)* is the central control program for the *vOS*. It is a process, which resides on a single workstation located anywhere within the network containing the *vOS* domain. The *vSM* manages individual workstations by interacting with a system service (a process) residing on each workstation named the *virtualizing EXecutive* (*vEX*). It displays the *vOS* system status and provides a command and control interface to each of the *vEX*s.

The *vEX* is located on each workstation that is a member of the *vOS* domain. It uses the application virtualizer, known as the *virtualizing INterceptor* (*vIN*), to capture and manage the individual workstation processes. The *vEX* consists of an application with a user interface that displays local system state and provides for local command and control activities.

The *vEX* uses the *vIN* to initiate and capture process data from the workstation applications. The *vIN* is a software module that is injected into a newly launched application. Once injected, it monitors the host application by intercepting its API calls to the underlying libraries. The injected code resides in the same address space as the application under its control. A simple command interface is integrated into the application through the insertion of a menu item into the existing application menu; otherwise, it is completely unseen by the user of the interactive application.

The structure for the *vOS* architecture is a tree with a single controlling element at the base (*vSM*), communicating with multiple individual systems through a module residing on each system (*vEX*) that in turn communicates with a module assigned to manage an individual application (*vIN*).

### 2.3 Technologies

Two key technologies enable the operation of the vOS. The first is the API/DLL Interception based on Richter's [4] description (also see section 5). APIs are intercepted by changing the addresses within the Import Address Table (IAT). The IAT contains a set of pointers filled in at runtime that resolve to the API fulfillment address in the appropriate DLL. This approach allows new code to be inserted between the application and the DLL. Once the API is intercepted, we have full access and control of the procedure call information.

The second technology is the creation of an abstraction layer between a running application and the underlying system through the creation and management of virtual handles. Virtual handles are values that are created to replace the original handles returned by the API from the system. The application saves and uses the virtual handles the same it would real handles. The virtual handles are connected to the real handles when the application makes requests to the system. This approach allows an application to be moved between computers, transparently, by remapping the unchanging virtual handles to real handles when required. The application remains unchanged and is not it aware of the underlying relationships.

State and handle data are collected, saved and managed within the *vIN*, reserving it in migration tables. The tables become a portion of the data copied and restored during the process migration operation described below.

## 3. Process Migration

A core activity for the *vOS* is the capturing of process API states, which enables the migration of processes between systems (see section 2 above). A prodromal condition for migration is the virtualization of the application handles and the capture of requisite state information. Once acquired, the *vOS* provides the mechanism for the migration. Collecting and managing the data for this migration requires that a variety of considerations and activities be coordinated between the *vOS* components. Specifically, the complexity of the program forms the guideline for the migration approach that can be employed. The following sections provide a review of the various data elements and activities managed by the migration system.

### 3.1 Migration Factors

It is at least theoretically possible to migrate or move any application between one or more systems. However, the complexity of the migration is dictated by a combination of several implementation factors associated with every process. Although these factors interact with each other, they represent unique and isolated activities in the migration algorithm. The following is a list of the key migration factors:

*Threading:* Each application consists of one or more threads. If each thread remains independent of any of the other threads in the process, then the handling of the state information has the same degree of complexity as a singular state process. Threads that dependent upon each other are considered *intertwined*. At the lower level, intertwining can take the form of shared global variables[2]. At the higher level it can take the form of complex signaling interrelations.

*User Interface* (UI): In general, the simpler the interface type, the less complex the task to migrate the state of the interface. For example, a text-based interface requires very little work to gather and restore as compared with a bit mapped interface.

*Input/Output* (IO)*:* If file IO is present, the complexity is represented by both the quantity and type. A single sequential file requires a file pointer, file name and physical

[2] Although Global Variables seem of little consequence to complexity within the same process address space, they introduce complexity when threads are placed onto separate machines.

address for a migration operation. A more complex example, such as one or more database transactions, requires that sufficient state knowledge be available to, either reconstruct an in-process transaction or to cause a transaction back-out and a subsequent restart after the migration.

*Network:* Network components are not considered movable or migratable, without the aid of some form of middleware or higher-level communication state management. Even using the approach pioneered in [5], more work is required and thus more complexity is introduced.

*Operating System:* The degree of homogeneity between the environments is crucial. Even though there is a general compatibility between the various versions of Microsoft Windows, migration between two versions, such as Win95 and Win2K, may not be possible for several reasons. Firstly, the content and implementation of the supported system APIs is different. Some of the APIs implemented in Win2K do not have direct correspondence with Win9x. In addition, syntactically equivalent APIs, can semantically differ. Even a migration between two more closely related systems, such as WinNT and Win2K, provokes implementation issues, when the semantic of an API varies subtly. Complexity increases between two systems, such as Linux and Windows, since there is little direct correspondence between their APIs and implementation. Complexity of this type is not directly solvable without the addition of either an API translation subsystem or a Windows/Linux API library.

*Role:* Today's systems usually consist of some form of either client/server or peer relationship with other systems. In either case, request and response messages are generated that affect the current state of the process, as well as the process' intermeshing with its environment. Migrating a process involved with commitments to or from adjacent systems creates additional complexity. The issues can involve, at the very least, the network relationships to one or more systems. They can also involve one or more stateful relationships where the state information is stored as a series of system table entries, such as are found in a Windows registry.

*Distribution:* Distribution is concerned with the aggregate state of a set of distributed applications. In a distributed system, complexity is gauged by the ability to identify and capture the state of the various application components of the process. Spatial arrangement comes into consideration here if the application must be co-local to some resource for operational or performance reasons. In this case, migrating the process may consist of more than just moving the state information.

*Platform:* Platform hardware homogeneity is a principle requirement for movement between platforms. In general, movement between two dissimilar hardware archi-

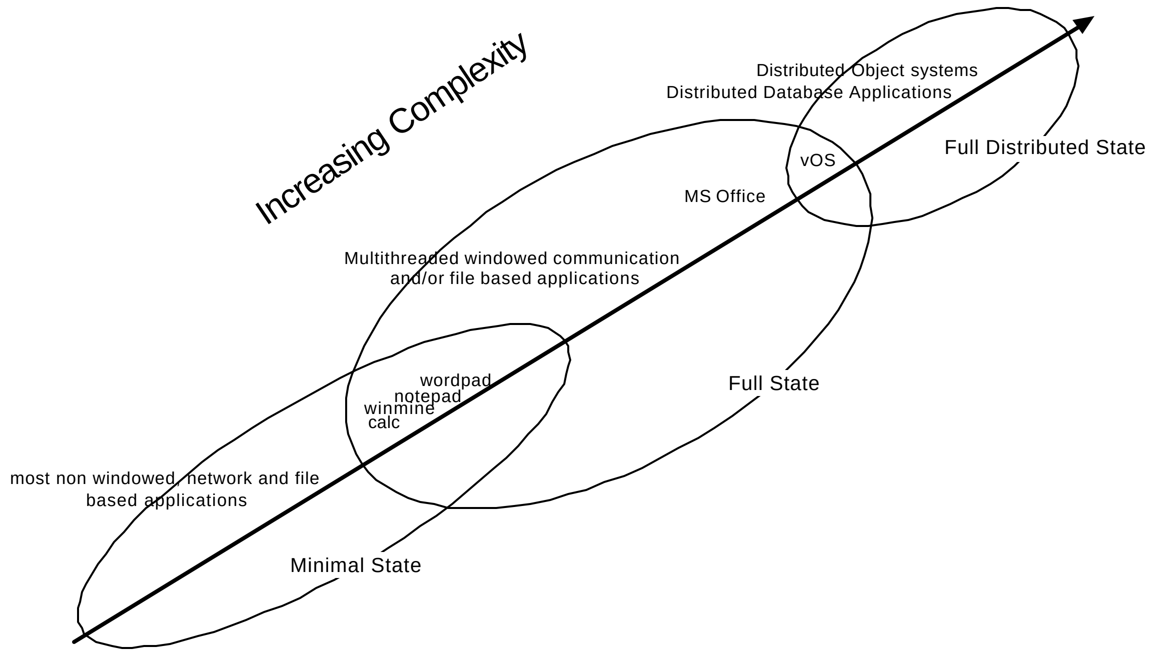
Increasing Complexity
Distributed Object systems
Distributed Database Applications
vOS
Full Distributed State
MS Office
Multithreaded windowed communication
and/or file based applications
wordpad
notepad
winmine
calc
Full State
most non windowed, network and file
based applications
Minimal State

**Figure 1: Migration Method Mapping**

tectures is not possible, without some form of emulation. Complexity is measured here by the degree with which corrections must be employed to assuage any system mismatches.

*Security:* Movement of any process requires that the security environs either be the same or modifications be made to adjust the settings to the new environment. If the process is using only the default security descriptors or the same descriptors in its operation, then complexity is eased considerably.

## 3.2 Migration Models

We discovered that the task of migrating a process is divisible into three methodologies based on the type of state information required (see figure 1). Simple processes require only a small or minimal amount of state information to be collected and restored; this approach is referred to as the Minimal State Migration Model. More complex processes involving network, file and threading require that a complete collection and restoration of process state be undertaken. This method is defined as the Full State Migration Model. Processes distributed over several machines and locations and effectively describing a distributed state model, must use the more complex Distributed State Migration Model which is not discussed in this paper.

### 3.2.1 Minimal State Migration

At the lowest level of migrational complexity is the partial-state migration method. Nasika and Heballalu [5,6] provided the pioneering work and Zhang, Khambatti and Dasgupta [7] extended their findings. Given an application of relatively low complexity, a certain minimal set of state elements combined with a restart/suspend technique is all that is required to correctly migrate the specific set of processes. This minimal set of elements consists of the ".data" area, heap allocation and handles. Tests show that these three components correctly recreate process state for applications containing one or more unrelated windows, one or more non-intertwined threads, no IO, no networking, same security, same platform, same OS, no role, no distribution and simple objects.

The .data, heap and handles are combined using a restart/suspend technique listed below. This technique succeeds by setting up the three migration components in such a way that they create a reproducible memory structure. This means that the address and content of the global data area, ".data" must be the same for each instantiation of the migrated process. This is in turn achieved by noting that in the Win2K environment, the ".data" area generally contains the uninitialized and initialized global and static variables for the application. However, these variables are not physically present in this region. The Windows implementation uses this area as storage for pointers into the heap where the actual data is placed. Thus, the heap must be included as part of the migration.

For the ".data" area to be consistent with respect to the handles, heap locations and values, two actions must take place. Firstly, the handles must be allocated such that the same virtual values are returned in the same order each time the application is restarted. Secondly, the memory allocations must be at the same locations and contain the same data for each program execution. These two constraints create a requirement for heap management to be performed as part of the migration system in order to ensure that both location and content are identical. This can be seen in the case of two dissimilarly configured systems where heap allocations of the native Windows routines are not guaranteed to generate the same addresses. Replacement heap functions provide this guarantee by using a fixed location in memory for the memory allocations, thus ensuring that memory remains identical for a given allocation ordering.

The algorithm that performs the migration consists of the following steps, which create the clone or copy:

1. Suspend the source process thread(s)
2. Write the handle, memory and .data
3. Terminate the thread

To restore the process the following steps are used:

1. Launch a new, suspended instance of the process
2. Inject the *vIN* and rebuild the handle tables
3. Resume the thread(s)
4. After thread(s) quiesces, suspend the thread(s)
5. Reapply, unapplied handles and memory
6. Restore the ".data" section
7. Resume the thread

### 3.2.2 Full State Migration

To migrate any non-distributed process requires the capture and restoration of the full process state. A benefit of this approach is that it provides support for a richer set of migration scenarios, containing one or more unrelated windows, one or more non-intertwined threads, IO, networking, differing security, same platform, same OS, differing role, no distribution and multiple objects. The state elements involved with this technique consist of the stacks, processor state, virtual handles, ".data" area, heaps and kernel objects.

In earlier work on process migration [2,3], it was believed that the processor state and heap data combined with handle virtualizing at the API/System DLL boundary layer was sufficient for reconstructing the process state. Further examination shows that crucial portions of the system state are missing, thus preventing effective reconstruction of the original environment. The missing information resides within the kernel and represents the process, thread, and windowing states. Reanimating a migrated application requires that kernel object states be recreated and reattached to their application level manifestations.

We determined that just restoring the stacks and threads is not sufficient to restart the windows engine. We examined an alternative approach to restoring the window state using window "sub classing" and "super classing" [8], which involves redirecting the window class procedure to an alternative user procedure. Although a limited success was achieved, it was not possible to fully reconstruct the pre-migration environment. For example, there is no method to reassociate the current queues with the premigration application state, so the subclassed window is stillborn on creation.

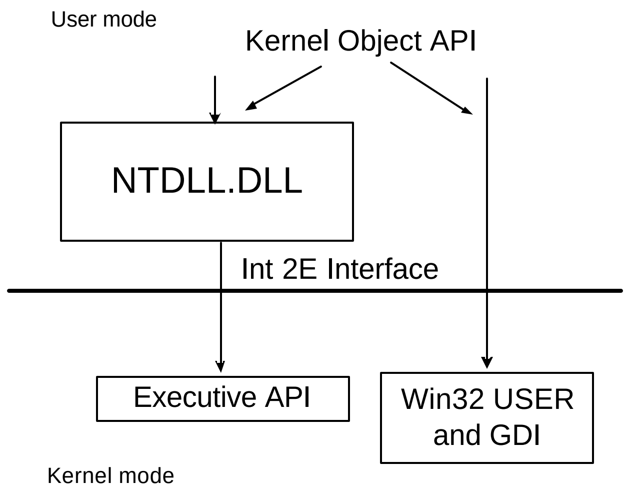

**Figure 2: User Kernel Call Interface**

To capture full process state requires that a virtualization occur between the user and kernel transition levels (see figure 2). Capturing the handle and state information at the kernel transition layer allows for virtualization of the object handles returned by the kernel, providing for later reconstruction of the underlying system state. The risk of using this approach is that the kernel API calls are not publicly documented and are subject to changes without notice. Therefore, changes to the operating system, including version and maintenance, may require alteration to the API capture routines

We observed that in many cases at the point of migration, the threads are within a system library which has in turn made a call to the kernel. Reestablishing these specific states is possible by using one of two approaches. The first approach is to reissue the kernel call after each of the kernel objects is recreated. The second is to fail the kernel call by allowing the call to return with a failure code, causing the system error handling routines to deal with the resulting failure. As a variation on this last theme, a special return value is used to alert the virtualizing code of the need for a restart, and then the call with the original captured data is reissued.

As in the Minimal State Model, heap management is also crucial. The heaps and .data relationships and locations must match between migrations. This means that the target heap memory must be reallocated at the same addresses without colliding with any preexistent heaps.

## 4. System Performance

The process migration mechanism has been implemented and tested as a part of a prototype *vOS* [10]. To perform process migration using the *vOS* as the control system requires that there be a minimum of two systems. The first system contains a *vEX* and the *vSM*. The second system contains a *vEX*. The operation of each of these components is described individually in section 2.

Testing was performed on a simple windows application known as *calc.exe* which is the stock Windows calculator accessory. The process of installing the interception APIs consists of injecting the target application with the *vIN* Dynamic Link Library code. The code, during its initialization, loads amongst other modules the hook library, which virtualizes the target APIs upon its instantiation. The time to inject the process is 526.2 ms and the time required to perform the complete *vIN* setup, including API interception, is 1108.7 ms.

The procedural logic to migrate a process is shared between the *vEX* and the *vIN*, which use overlapping execution with system signal synchronization. To develop a performance picture for the process migration, it is best to measure the lower level procedures that actually perform the data collection and restoration activities.

To migrate a process, the *vOS* library routines perform the complete handle, heap and .data collection, formatting and migration file build function. The overall clone build process time is 2,385.8 ms, which includes the file write time. Contained in this time are the following elements:

| | |
|---|---|
| • Handle/Memory collect and format | 68.7 ms |
| • .data collect and format | 69.9 ms |
| • Heap compaction | 192.6 ms |
| • Heap collect and format | 264.5 ms |

The process is restored as part of the *vIN* initialization routine. Since this is also a complete restart of the process, the process is injected with the *vIN*, taking 362.6 ms, which is less than the previously measured time due to the *vEX*'s direct involvement with the process restore.

The *vEX* acts to directly control the restore process, requiring less overhead than the normal system injection processing. The overall restore time is 2,172.0 ms. This time is less than the original migration build time, in part due to the use of the Minimal State migration model (section 3.2.1) which does not require the heaps to be restored. After the state information is restored and the application is allowed to initialize to the point of displaying the window, any handle table and memory table values that remain are replayed and new real values are recreated[3]. This procedure brings the application back to the same state that existed at the time of migration.

The timings for the restore elements are:

| | |
|---|---|
| • Handle/Memory restore | 337.3 ms |
| • .data restore | 34.1 ms |
| • Handle table replay | 17.0 ms |
| • Memory table replay | 35.4 ms |

The total *vIN* setup time required to support the restoration is 1616.1 ms. This time is slightly higher than the non-migration time, but is to be expected due to the additional procedure time associated with restoring a process.

## 5. Related Work

There are two fundamental sets of technology related to this work. The first is the API injection and interception methods. The second is the system structure supporting the injection and interception.

**Mediating Connectors** [11] was developed by the USC Information Sciences Institute and consists of a library and run time applications. It wraps the DLL API with code developed by the user and ensures that the new code is invoked when the wrapped API is referenced in the application. Mediating Connectors is also known as NT Wrappers. We originally considered incorporating this technology into the *vOS*. However, its main focus is on relatively standard applications which rendered it programmatically heavy and inflexible. The *vOS* required more system level control and access to the source modules and thus a facility that was more lightweight.

**InjectLib** [4] is one of several prescribed methods used for injecting[4] application code into an active process from a separate process on the same machine. Process *A* injects a stub routine *S* into process *B*, using standard Windows system calls. *S* is a separate thread that in turn loads a DLL module *D*, specified by process *A*, as part of the injection. *D*'s initialization code is executed after it is loaded, thus providing the base for further interaction with process *B*. This approach is incorporated into the *vOS*.

**Detours** [12] from Microsoft Research is an API interception application library using 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 access to the original function is provided using a *trampoline* function. Rewriting the application code creates several potential security, portability and compatibility issues.

**COP** [13] is a collaboration between Microsoft Research and the University of Rochester using Detours. It is Microsoft Foundation Class (MFC) and Common Object Model (COM) oriented, building and wrapping components around the Win32 API. Applications that do not use COM are not candidates for inclusion in this technology.

**NT-SwiFT** [14] also known in its first production release from Lucent Technologies as *SwiFT for Windows NT*. The system is designed to support client/server architectures and provide high availability and reliability. In the role of fault tolerance and high availability, it is capable of migrating applications between systems and restarting them in the event of a failure. The facilities provided can be used standalone or can be integrated into a complex set of system components. It is assumed that server application can and should be modified to incorporate the *SwiFT* capabilities and that client applications do not necessarily need to be modified, although *SwiFT* provides client level API support. *SwiFT* performs API interception of a limited number of API's and DLL's.

**Transparent Checkpoint Facility On NT** [15] was specifically designed to checkpoint long running engineering applications. It performs API interception the same way as the *vOS* by capturing system and application state while the application executes then replaying 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 alteration, 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 bypassing the IAT may not work correctly and multiple interacting processes are considered too complex to handle. It does not appear to handle windows or non-file oriented networking.

## 6. Summary

In this paper, we have presented experimental process migration work using the virtualizing Operating System or *vOS*. The *vOS* is the main platform implementation of

[3] Values remain after the replay. They are predominantly handles created by keystroke activites not duplicated during the restore procedure.
[4] When code is placed into an existing process address space from an adjacent address space, the new code is termed as "injected".

the Computing Communities project and forms the basis for implementing the core technologies. It is distributed in nature and is integrated into the existing Windows 2000 environment without altering either the applications or the operating system.

The main focus of this research is the identification and implementation of migration technologies using the *vOS* as a base. We divided the applications into three fundamental categories based on degree of complexity. Three methods were defined for use depending on the application complexity. The first method is the Minimal State and handles simple or non-complex applications. The second method, Full State, handles more complex non-distributed applications. The third method is the Distributed State method, which handles applications spread over a network.

This work has described the process and performance of the Minimal State method. After conducting further research, we find that the Full State method it is both feasible and relatively straightforward to implement into the *vOS*, using the technologies and techniques we have perfected. The bulk of the work of implementing the Full State process migration model is not the virtualization and state capture issues, since we have implemented those technologies with the Minimal State model. The difficulty in implementing the Full State model is in integrating a reasonable subset of API hooks from the over 1700 published APIs implemented in the system Dynamic-Link Libraries[5].

## 7. 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] Tom Boyd and Partha Dasgupta, *Virtualizing Operating Systems for Seamless Distributed Environments,* in Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, vol. 2, November 2000, pp. 735-740.

[3] Tom Boyd and Partha Dasgupta, *Injecting Distributed Capabilities into Legacy Applications Through Cloning and Virtualization,* in the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, vol. 3, June 2000, pp. 1431-1437.

[4] Jeffrey Richter, *Applications for Windows, Fourth Edition,* Microsoft Press, 1999.

[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] Shu Zhang, Mujtaba Khambatti and Partha Dasgupta, *Process Migration through Virtualization in a Computing Community,* 13th IASTED International Conference on Parallel and Distributed Computing Systems, August 2001.

[8] *MSDN Library – January 2001*, Microsoft Corporation, 2001.

[9] K. Mani Chandy and Leslie Lamport, *Distributed Snapshots: Determining Global States of Distributed Systems,* ACM Transactions on Computing Systems, vol. 3, no. 1, pp. 63-75, February 1985.

[10] Tom Boyd, Virtualizing Operating Systems for Distributed Services on Networked Workstations, PhD thesis, Arizona State University, Tempe, AZ, December 2001.

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

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

[13] Robert J. Stets, Galen C. Hunt, Michael L. Scott, *Component-based APIs: A Versioning and Distributed Resource Solution"*, IEEE Computer, vol. 32, no. 7, July 1999.

[14] 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.

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

Sponsor Acknowledgment: Effort sponsored by the Defense Advanced Research Projects Agency, under agreement numbers F30602-99-1-0517 and N66001-00-1-8920. 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, or the U.S. Government.

[5] The versions of NTDLL.DLL and GDI.DLL current to this writing contain 1187 and 543 API entries respectively for a total of 1730.