Transparent Migration of Distributed Communicating Processes [1]

Ravikanth Nasika Tandem Corporation
Partha Dasgupta Arizona State University

Abstract

A Computing Community is a group of cooperating machines that behave like a single system and runs all general-purpose applications—without any modifications to the shrink-wrapped binary applications or the operating system. In order to realize such a system, we inject a wrapper DLL into an application at runtime that manages the execution of the application and endows it with features such as virtualization and mobility.

This paper describes the concept of virtualization, and the mechanism of injection and the implementation of a wrapper DLL. We focus on one kind of applications, those that use sockets to communicate with other processes. We show how these processes can migrate between machines without disrupting the socket communications. We have implemented the software that needs to be injected into the application to enable this feature. Handling more application types is part of the continued research in the Computing Communities project.

1. Introduction

The impact of the "Networks of Workstations (NOW)" approaches and the "cluster" approaches for distributed computing has had limited impact on the general-purpose computing arena. The NOW approach has resulted in a plethora of parallel computing platforms (such as PVM [12], MPI [13], Calypso [10], Linda [6], Treadmarks [1], Brazos [25] and so on). The clustering approaches as been successful in a few special application areas such as highly reliable file and database and web services (notably from Sun, Tandem, IBM and Microsoft).

The promise of distributed computing for general-purpose computations was a major thrust in the development of Distributed Operating Systems in the mid-1980s. Operating systems such as Amoeba [27], Mach [21], Clouds [7], Chorus [22], and so on failed to bring the power of distributed general purpose computing to the desktop. In retrospect, the shortcoming of these systems resulted from the *application development barrier*.

The application development barrier is the void of applications for a new platform. If we design a new distributed operating system, we endow it with a lot of novel features, but this requires a redevelopment or at least a rewriting of all the applications that have to run on the system, and often such a task is infeasible. The application development barrier is further complicated by the need to use newer and ever-expanding array of APIs needed for implementing the enhanced features. Currently, API bloat is causing many of the new features invented, to be ignored by the applications.

Our research is targeted to bridging this gap to enable all existing (legacy) shrink-wrapped applications to be executable on a new distributed platform, where the distributed platform supports all the abilities of a true distributed system, such as distributed scheduling, process mobility, failure masking, load balancing and so on. We want to preserve old applications, old APIs and make these systems get new features without the use of any new APIs or programming strategies. The system that we are attempting to build is an integrated distributed computing platform that we call as a "*Computing Community*" (see section 1.1).

The basic mechanism we use to build computing communities is the *unobtrusive injection* of new functionalities 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 the next subsection we discuss the features and approach of the Computing Communities project. Section 2 illustrates how API Interception is achieved. Section 3 provides details about our process migration facility and section 4 describes how processes with network connection are migrated (while preserving the connections). Section 5 discusses related work. Our research uses the Windows NT operating system, but the techniques are applicable to other operating systems such as Unix or Linux.

[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.

1.1 Computing Communities

Our research is part of a larger project called "*Computing Communities*" (or *CC*) [9]. 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 migration of processes, applications and resource allocations between systems—all transparently.

The computers participating in the *CC* utilize a standard operating system and run shrink-wrapped 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". Virtualization 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. The application uses 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. Some of the advantages of virtualization of resources are:

- A critical application running on one machine can be moved to a new machine, if the first machine has to be shutdown. This will be useful to perform maintenance without service interruption (*reassignment of a physical CPU for the virtual CPU*).
- Having multiple physical resources for a virtual resource delivers many new capabilities ranging from replicating the application displays on multiple monitors to replicating processes on multiple machines for fault tolerance. The application does not need to implement these mechanisms (*reassignment of, possible multiple, physical resources to virtual resources*).
- In a CC, the scheduler can schedule the processes on any machine in the CC independent of what resources are used on the machine when they are started. This can be used to achieve the required quality of service.
- The users can change their virtual "home" machines at will, even for applications that are currently running. Thus a login session can be detached from one machine and reattached to another machine, dynamically. This is the ultimate mobile computing scenario.

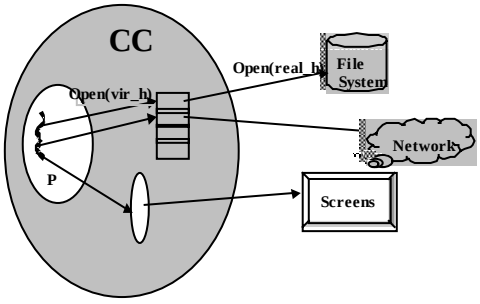

Figure 1: Translating virtual handles to physical handles to enable virtualization for a process P.

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 1). 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.

This paper explains a mechanism to virtualize the network connections used by a process i.e., the process can move from one machine to another, but still sees the same network connections.

2. API Interception

The virtualizing Operating System (*vOS*) sits between the applications and operating system (Figure 2). The *vOS* should be built without changing the source or binary code of the applications so that all applications utilize these advantages of the *vOS*. The unobtrusive injection of the *vOS* functionality can be done by intercepting calls made from the application to the underlying runtime system and reinterpreting the call. This is called *API interception* [4, 16]. API interception has been used to implement transparent services such as file compression (Stacker) and extensible file systems (Ufo [2]).

On Windows NT, an application requests the OS services by making Win32 API calls. All the necessary API calls made by the application should be intercepted once the application starts running. To achieve this, we have

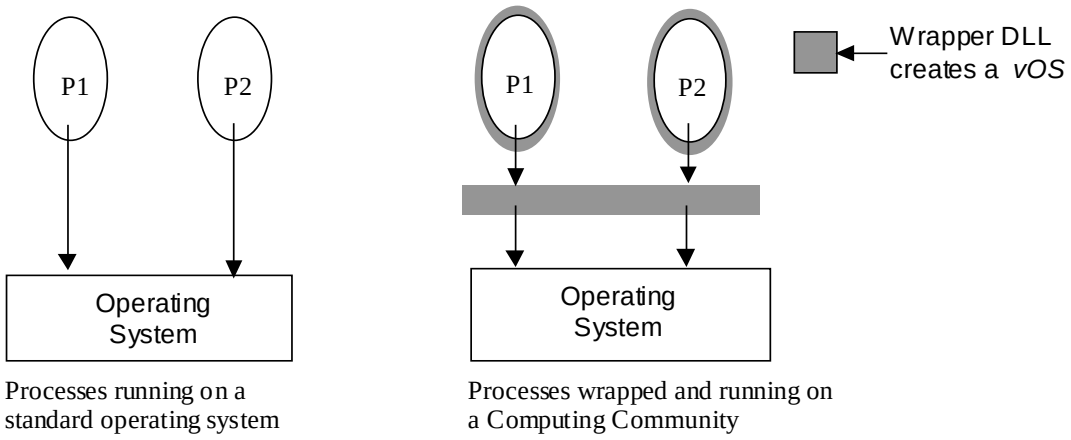

Figure 2: Using API interception to create wrappers that implement the CC

built "*Wrapper APIs*". The wrapper API is a proxy of the Win32 API that we want to intercept. Whenever an application program calls a Win32 API routine, if there exists a corresponding wrapper API, the wrapper API is automatically called. The wrapper API can implement a variant of the original API or pass the call to the original API, or reinterpret the API call and modify it to execute something quite different, in conjunction with calls made to several of the originals APIs. The addresses of Win32 APIs are stored in the import address table (IAT) in the process's virtual address space. When a call is made to a Win32 API, it is redirected to the IAT from which another jump occurs to the actual Win32 API. The addresses in the IAT are changed to the addresses of wrapper API so that when a call is made to an API, the wrapper API is called.

2.1 Wrapper Implementation

The approach to implement the wrapper API is to (1) map the wrapper APIs' code into the process's virtual address space and (2) change the import address table addresses to the wrapper API addresses. These two steps can be achieved by injecting a DLL into the process's virtual address space using the methods described in [23]. In order to inject a DLL, a stub routine is injected (or copied) into the process address space and then this code is run in the process context using the remote thread mechanism explained in [23]. This code loads the DLL that contains all wrapper APIs. When the DLL is loaded, the DLL initialization function (DLLMain()) is called, and this routine changes the IAT to point to the wrapper API addresses. Next time when the Win32 API is called, wrapper API is activated instead of the original Win32 API.

3. Process Migration

Process Migration is the relocation of a process from one machine to another machine. Process migration has been heavily studied and implemented in the past without any major impact to distributed computing. Process migration has typically been used for load balancing in parallel processing systems and includes systems such as MPVM [5] and DynamicPVM [8], and Chime [27]. Operating systems that support process migration include Chorus [22], MOSIX [3], Amoeba [27], and Sprite [11]. However in these systems, the applications have restrictions, or have to be written for the particular OS, or have to be recompiled or re-linked.

Past attempts at process migration have been limited to processes without any I/O connections or GUI handling. The lack of impact has been largely due to the fact that regular unmodified programs, which have external dependencies, are generally considered not migratable.

In our research, migrating processes forms one of the core features that is necessary to realize the power of a *CC* [14, 18]. Hence the ability to migrate real processes—i.e. processes running a standard unmodified binary of a commercial application is essential. In this paper we focus on the migration of processes with network connections [18].

To migrate a process, its execution should be stopped, and its state should be captured. Typical process state consists of: thread contexts, threads' stacks, text region, data region (static memory), dynamic memory and system state. The thread context consists of all the register values like program counter, stack pointer etc. The text region is the code region in the process's virtual address space that is typically memory mapped from the executable from the disk. The data region consists of global and static variables used by the program along with other information. Heaps contain the dynamic memory allocated for the application. System state consists of the process information maintained by the operating system.

A process may use files, network connections and inter-process communication. When the process is migrated, its state should be restored so that process starts executing where it was stopped before on the source machine. Of course, from the Operating System's point of view the process is not migrated at all – we simply create a new

process and manipulate its state so that it will be identical to the original process.

3.1 Capturing And Restoring State

The capture of the state of a running process can be achieved if the process was injected with a wrapper when the process was started. At load time, the process is started in a suspended state and a wrapper DLL is injected into the process. Then a thread is started inside DLL initialization routine (DllMain) in the wrapper DLL. This thread (called the *control thread*) waits for an external request to capture state (Figure 3).

As the process executes, the wrapper DLL monitors the external effects of the application and records such events such as creation of sockets, opening of files and creation of GUI objects. When a process creates (or attaches to) a resource, the handle returned by the operating system is replaced by a "virtual handle" by the wrapper DLL. The virtual handle is a system-wide unique resource identifier. The application uses only the virtual handles and the wrapper DLL performs the translation.

When the process is to be migrated from one machine (say source machine) to another machine (say target machine), the control thread saves the process state and sends this information to the target machine. On the target machine, the application process is started in the suspend mode and the wrapper DLL is injected and control thread is created. The control thread on the target machine now restores the process state and resumes the process. The next subsections explain how various parts of the process state are captured on the source machine and restored on the target machine.

3.1.1 Thread Contexts:

When the process is to be migrated, the control thread gets the main thread's context using GetThreadContext() API on the sources machine. The control thread restores the thread's context using SetThreadContext() API on the target machine.

3.1.2 Data Region

The Data Region consists of .rdata, .idata, .edata and .data sections. The .rdata section consists of read only data like constant strings and constant integers etc. The .idata section contains the import data including the import directory. The .edata section consists of the export data for an application or DLL. All the remaining global variables are stored in the .data section. There is no need to store the .idata or .edata or .rdata sections as they are restored from the executable automatically when the process is restarted on the new machine. The starting address of the data region can be found from the portable executable file format [19] that is mapped to the process's virtual address space and the size of the region is found by calling VirtualQuery() on that starting address. On the target machine, the control thread restores the data region at the same address.

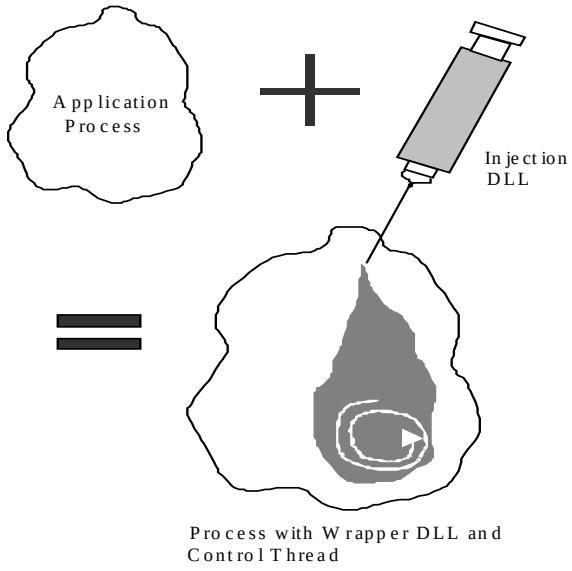

Figure 3: Injecting a processes with wrapper and control thread.

3.1.3 Dynamic Memory

On Windows NT, the dynamic memory can be allocated using heaps or by creating virtual memory using VirtualAlloc(). Information about the heaps like the number of heaps and addresses of heaps can be found from Process Environment Block (PEB). Each VirtualAlloc() call is intercepted and information about the memory regions allocated is saved on the source machine and restored exactly at the same locations in the process's virtual address space on the target machine.

3.1.4 System State

Most of the information about the system state is stored in the kernel space and cannot be accessed from the user space. There are a few Win32 APIs that an application can use in order to retrieve this information. These APIs can be wrapped with wrapper APIs using the DLL injection mechanism described in previous sections. The wrapper APIs make the application see the same system state on the target machine after the process is migrated. All the handles seen by the application are virtualized i.e., the application sees the same virtual handles though the physical handles on the target machine may be changed.

A process may use files, sockets, inter-process communication and other graphical interfaces. When the process is migrated, all the file and socket handles that are opened by the process are invalid on the new machine. Virtualizing the handles solves this problem. When the process creates either a file or a socket using a Win32 API, a virtual handle is returned to the application. The

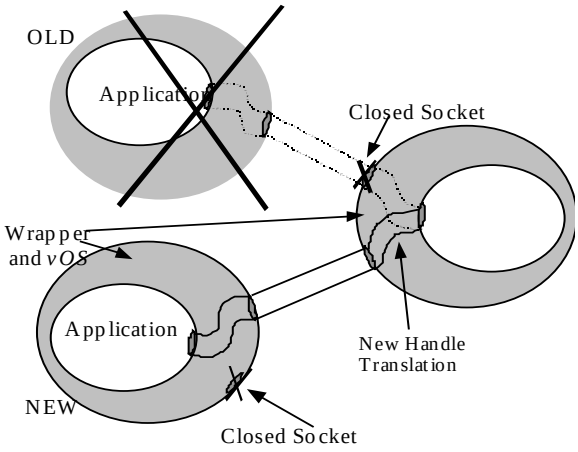

Figure 4: Connecting the sockets after migrating a process.

wrapper APIs that are part of the *vOS* dynamically map the virtual handles to the physical handles. The next section explains migration of processes with (TCP/IP) network connections using virtualization.

4. Migrating Connected Processes

Generally, network connections are created as follows: One process acts as a "server", and another process as a "client". The server creates a socket and binds to a well-known port on a particular machine. Then, it waits for a connection from the client. The client knows the IP address and port number of the server. The client first creates a socket and then connects its socket to the socket created by the server.

When a process has to migrate to another machine, it closes all the connections with other processes and when it is migrated to the new machine, it reestablishes the connections. From then onwards, communication occurs as if the process has not migrated. The next subsections explain how this can be done transparently without knowledge of applications.

The entire socket APIs are wrapped with wrapper APIs. When the application calls a socket API, the corresponding wrapper API is called. These wrapper APIs return the virtual handles to the application. As in the case of saving process state, the control thread also handles the network connection migration.

4.1 Migration of a Client Process

When the client has to migrate, the control thread saves the information about the sockets and sends a message to the server to close the connection to the control thread on the server. When it receives the OK message from the control thread on the server, it closes the connections. The control thread on the server also closes the connection and waits for a message from the client process on the new machine.

When the client process is restarted on the target machine, the control thread restores the process state. The control thread reads the socket information from the checkpoint file and gets the server IP address and port number. The control thread sends a message to the socket thread on the server machine that it is ready to reopen the connections.

The control thread makes a connection to the server and saves the new socket handle. However, the application still sees the same virtual socket handle and the mapping between the virtual socket handles to physical handles is done in the wrapper APIs (Figure 4).

4.2 Migration of a Server Process

The same method is followed when the server is migrated from one machine to another except that when the server is migrated to the target machine, it sends the IP address of the target machine to the client's control thread. It allows the client to know the server's new IP address and reestablish all the connections. It is not necessary when the client migrates, because the client has to know the server's IP address in order to establish a connection whereas server need not know the client's IP address.

The above scheme can be extended to work when the application to be migrated is not talking to a process that has a wrapper DLL (a process outside the CC). In this case, all communication between wrapped applications and non-wrapped applications has to go through a gateway (or proxy). The connections between the applications and the proxy can be renegotiated when they migrate, but since the proxy does not migrate, the proxy to non-wrapped application connection remains the same.

5. Related Work

There are several existing solutions to process migration problem on UNIX like MPVM [5], Condor [17], LibChkPt [20], MOSIX [3], and Sprite [11]. To our knowledge, the systems that provide process migration on Windows NT are NT-SwiFT [15] at Bell Labs and a checkpoint facility [24] developed at Intel, Israel. The following sections explain the related work.

5.1 MPVM

Parallel Virtual Machine (PVM) is a software system that allows a network of workstations to be programmed as a single computational resource. Migratable PVM (MPVM) [5] is an extension of PVM that supports transparent process migration. MPVM allows parts of a parallel computation to be suspended and later resumed on other workstations by migrating the process state from one machine to another. Transparency is ensured by modi-

fying the PVM libraries and daemons and by providing wrapper functions to some system calls. This mechanism is implemented at the user level. Some of the limitations are: The programmer has to create the executables that are statically linked with the libraries. This system is for only PVM computations and not for general purpose processes.

5.2 Condor

Condor [17] is a distributed batch processing system for UNIX. It schedules jobs in a network of workstations. It transparently checkpoints the process state to a file and restarts a process possibly on a different machine. Programs are re-linked (but not re-compiled) to include the checkpoint libraries. It is implemented at the user level. Condor has the following limitations: Unable to migrate one or more members of a set of communicating processes. Processes can not execute fork() or exec() or communicate with other processes via signals, sockets, pipes, files, or any other means. Condor checkpointing code must be linked in with the user's code and it does not work for users of third party software who do not have the access to the source.

5.3 Libckpt

Libckpt [20] is a portable transparent checkpointing library on UNIX. It checkpoints the process state, i.e. it saves the process state to a file and recovers the process state after a failure. It uses transparent incremental and copy on write checkpointing. It also implements user-directed checkpointing to improve performance based on the assumption that a little user input to the checkpointer can result in a large performance payoff. To use libckpt, the developer has to change one line of his source code and recompile with libckpt library. It runs at the user level. Here are some of its limitations: It does not support checkpointing of communicating processes. User has to change the source code and recompile the application.

5.4 Sprite

Sprite [11] provides transparent process migration to allow load sharing by using idle workstations. It is implemented at the kernel level while providing a UNIX like system call interface. In Sprite, each process appears to run on a single host known as host node throughout its lifetime, but it may execute physically on a different machine. The kernel distinguishes between location–dependent and location-independent calls. The kernel forwards location-dependent system calls of a foreign process to its home node. Whenever modules that implement migration need to be changed, many parts of the kernel have to be changed.

5.5 Checkpoint facility on NT

This is one of the very few checkpoint mechanisms on Windows NT. It has been developed at Intel, Israel. It implements a checkpoint facility [24], a general-purpose library that can be linked and used with any application transparently. This system is able to checkpoint the processes by redirecting the Win32 API calls and saving the data segments, thread execution context and stack segments. Here are some of the limitations of this system: The user has to relink the application with the checkpoint DLL. It does not checkpoint and migrate communicating processes.

5.6 NT-SwiFT

Software Implemented Fault Tolerance on Windows NT (NT-SwiFT [15]) is a set of components that facilitates building fault tolerant and highly available applications on Windows NT. It checkpoints data segment, communication channels, contexts of threads, stacks etc. NT-SwiFT provides detection of failure of a client program; At fault recovery, the client program is restarted and communication channels to server programs are reestablished. This system is not general enough to be used for most applications, and is targeted towards telecommunication industry, especially client server programs for fault tolerant services. When a process is migrated to a new machine, handles used by the old process have to be identical to those used by the new process—and the intercept routine repeatedly allocates handles until the returned handle is identical to the old handle. This may be an expensive operation.

6. Summary

Most systems that provide process migration, either change the kernel, or application code. Our system implements the process migration mechanism without any modifications to the application or the OS, by using the Win32 API interception mechanism. Any process that is not inherently built to migrate can migrate to a different machine using this mechanism. This paper only describes communicating processes and [14] extends it to GUI programs. The API interception technique can be used for general applications (work in progress).

As described earlier, our system provides a new mechanism that virtualizes the resources used by any application. By doing so, we can map the resources dynamically without the knowledge of the application.

7. References

[1] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer,* December 1995.

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser and C. J. Scheiman. Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System, *ACM Transactions on Computer Systems*, August 1998.

[3] A. Barak, S. Guday and R. G. Wheeler. The MOSIX Distributed Operating System, *Lecture Notes in Computer Science*, Vol. 672, Springer, 1993.

[4] R. M. Balzer and N. M. Goldman. Mediating Connectors. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop*, Austin, Texas, pp. 73-7, June 1999.

[5] J. Casas, D. L. Clark, R. Konoru, S. W. Otto, R. M. Prouty and J. Walpole. MPVM: A Migration Transparent Version of PVM, *Computing Systems: The Journal of the USENIX Association*, 8(2), Spring 1995.

[6] N. Carriero and D.Gelernter. Linda in Context. *Comm. of ACM*, 32, 1989.

[7] Dasgupta, P., LeBlanc Jr., R. J., Ahamad, M. and Ramachandran, The Clouds Distributed Operating System. In IEEE Computer, Nov. 1991.

[8] L. Dikken, F. van der Linden, J. J. J. Vesseur and P. M. A. Sloot. DynamicPVM: Dynamic Load Balancing on Parallel Systems, In W. Gentzsch and U. Harms, editors, *High Performance Computing and Networking*, pp. 273-277, April 1994, Springer Verlag, LNCS 797.

[9] 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.

[10] P. Dasgupta, Z. M. Kedem and M. O. Rabin. Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, 1995.

[11] F. Douglas and J. Outerhout. Process Migration in the Sprite Operating System. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pp. 18-25, September 1987.

[12] Al. Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, 1994.

[13] W. Gropp, E. Lusk and A. Skjellum. Using MPI Portable Parallel Programming with the Message Passing Interface. *MIT Press, 1994*, ISBN 0-262-57104-8.

[14] R. Hebbalalu. *File I/O And Graphical I/O Handling for Nomadic Processes on Windows NT*, MS Thesis, 1999. Arizona State University.

[15] Y. Huang, P. E. Chung, C. Kintala, D. Liang and C. Wang, NT-SwiFT: Software Implemented Fault Tolerance for Windows NT, *2nd USENIX Windows NT Symposium*, July 1998.

[16] [16] G. Hunt and D. Brubacher. Detours: *Binary Interception of Win32 Functions*, Microsoft Research Technical Report, MSR-TR-98-33, February 1999.

[17] M. Litzkow, M. Livny and M. Mutka. Condor—A Hunter of Idle Workstations, *8th International Conference on Distributed Computing Systems*, 1988.

[18] R. Nasika. *Migration Of Communicating Processes via API Interception*, MS Thesis, 1999. Arizona State University.

[19] M. Pietrek. *Peering Inside the PE: A tour of the Win32 Portable Executable File Format.* Microsoft MSDN Library, March 1994.

[20] [20] J. S. Plank, M. Beck and G. Kingsley. Libckpt: Transparent Checkpointing under UNIX. In *Proceedings USENIX* Winter 1995, New Orleans, Lousiana, January 1995.

[21] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alesandro Forin, David Golub, Michael B. Jones, Mach: A System Software Kernel, *Proceedings of the 1989 IEEE International Conference COMPCON.*

[22] M. Rozier, V. Abrossimov, F. Armand, M. Gien, M. Guillemont, F. Hermann and C. Kaiser, Chorus (Overview of the Chorus Distributed Operating System), *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992.

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

[24] J. Srouji, P. Schuster, M. Bach and Y. Kuzmin. A Transparent Checkpoint Facility On NT. In *Proceedings of 2nd USENIX Windows NT Symposium*, August 3-4 1998.

[25] E. Speight and J. K. Bennett. *Brazos: A Third Generation DSM System*, The USENIX Windows NT Workshop, 1997.

[26] S. Sardesai, D. McLaughlin and P. Dasgupta, Distributed Cactus Stacks: Runtime Stack-Sharing Support for Distributed Parallel Programs, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, July 1998.

[27] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen and G. van Rossum. Experiences with the Amoeba Distributed Operating System, *Communications of the ACM*, 33(12), 1990.

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.