Processes Migration through Virtualization in a Computing Community

Shu Zhang, Mujtaba Khambatti, Partha Dasgupta
Arizona State University
Tempe, AZ 85287-5406
U.S.A.

ABSTRACT

The Computing Communities project uses an innovative approach to integrate the ubiquitous desktop with an underlying distributed system. The approach involves unobtrusive modification of functionality of existing systems by decoupling the application process from the operating system. Using this, we are building an integrated distributed computing platform. Process migration is an underlying mechanism that is key to enabling the Computing Communities framework.

We migrate regular, general-purpose computations, running shrink-wrapped binaries. By augmenting the decoupling of applications with techniques that checkpoint and restore the state of a process, we are able to migrate a process within our distributed environment. In this paper we outline our experiences in migrating Win32 process running over Windows 2000.

KEY WORDS: Parallel/distributed computing systems, API Interception, Process Migration.

1. INTRODUCTION

Over the past two decades, research in the distributed systems arena, has yielded a set of parallel processing platforms, (such as PVM [1], MPI [2], Calypso [3], Linda [4], Treadmarks [5], Brazos [6] and so on) and many distributed Operating Systems (such as Amoeba [7], Mach [8], Clouds [9], Chorus [10], and so on). However, this research failed to bring the power of distributed general-purpose computation to the desktop. We believe that this can be attributed to three major reasons.

First, enhancements to existing system capabilities can potentially invoke system-wide modifications that can become expensive [11]. This increases cost of development and has therefore not been popular amongst researchers. Second, the void of applications for a new platform, also called the *application development barrier* [12] makes such platforms unattractive. Finally, legacy applications need to be rewritten in order to use the features of the new platform [11]. Again this leads to the increase in the cost of development and deployment.

We feel that the solution to this problem calls for extending current desktop operating system technology to provide the attributes of a distributed system. Further, existing shrink-wrapped applications must be enabled to execute over this distributed environment and benefit from the platform's ability to perform distributed scheduling, process migration, failure masking, load balancing and so on. 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). Our efforts demonstrate the power of unobtrusive modification of functionality of existing systems without any change to the binaries of the base operating system or its application base.

In this paper, we describe how such unobtrusive modification of functionality can be achieved by using virtualization [13, 14] and well-known API Interception technology [15] to decouple the application process from the operating system. We further illustrate how it can be used to allow processes to be migrated within a computing community. The processes in our system would attain attributes such as mobility, collaborative work, distributed systems management, automatic reconfiguration, and fault tolerance [16]. Finally, we outline recent experiences with migrating Win32 processes running over Windows 2000, and identify several research issues that require further exploration.

1.1 COMPUTING COMMUNITIES

Our research is part of a larger project called "*Computing Communities*" (or *CC*) [13]. 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 requires 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 Syste*m" or *vO*S. The main theme in the *vOS* is, of course "virtualization".

1.2 VIRTUALIZATION

Virtualization is the decoupling of the application process from its physical environment [12, 11]. That is, a process runs in a virtual environment with connections to a virtual screen and virtual keyboard. The application uses virtual files, virtual network connections, and other virtual resources. For every virtual resource that the process needs there exists a mapping, provided by the virtual environment, to a physical resource of the operating system. When the application attempts to access a virtual resource through a system call, the virtual environment intercepts that call and changes the parameters of the system call to access the actual physical resource. This enables the application to use remote resources as though they were local and change the mapping between the virtual resource and physical resource dynamically.

1.3 API INTERCEPTION

The mechanism used by the virtual environment to intercept system calls is known as *API Interception* [13]. In the Windows 2000 Dynamic Linked Library (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. By modifying the addresses contained in the IAT, the application call is redirected to an alternate API entry point. Inserting code at that entry point introduces new functionality that creates the opportunity to track the request and use of resources by the process. This method is further described in [15]

The virtual environment begins to intercept system API calls at the very beginning of the process to guarantee that all the resources used by the process are virtual resources and that the process is completely decoupled from the physical environment. Therefore, at the point of creation of the process, the API Interception DLL is injected into the process space. Thus, we achieve unobtrusive modification of functionality of the existing system.

In this paper we describe the mechanism by which virtualization and API Interception can allow processes to migrate within a Computing Community. We further illustrate the techniques that are necessary to checkpoint and restore the process state in order that the process can start on the new machine in the same state as it was before migration.

2. MOTIVATION

Traditionally, process migration mechanisms have been used to load balance processors in a distributed system and approaches for supporting them transparent to the application have required extensive kernel support. We have described the two techniques that were most commonly used to provide process migration capability.

- Many programming systems and batch environments that rely on a checkpoint/restart strategy for spreading computations across a network of machines link the process to a user-level migration library, which handles process state maintenance and migration facilities. However, this method restricts the process from being involved in GUI interactions, open network connections, or accessing file systems. Example systems include Condor [17], Emerald [18], Charlotte [19], and Chime [14, 20] programming systems, as well as migration-capable extensions of PVM [21] such as MPVM [22] and DynamicPVM [23].

- Most distributed operating systems with support for process migration such as Chorus [10], MOSIX [24], Amoeba [7], SPIN [25], and Sprite [26] relocate the inner core of the process along with related kernel state using system mechanisms. They optionally leave behind a proxy on the source site to handle any "residual dependencies" (e.g. active network connections). Such a strategy can migrate any general process, but incurs significant run-time overheads in addition to operating system complexity and maintainability costs.

Despite the large number of process migration prototypes described in literature, relatively few applications have been shown to benefit from these mechanisms. Several of the distributed systems referred to above have used process migration for load-balancing jobs in a network, and parallelizing coarse-grained applications certain scientific computations, with results in literature reporting good throughput improvements.

Unfortunately, these applications are no longer viewed as compelling, particularly given the ready availability of small- to moderate-sized SMPs. Nonetheless, we attempted to employ process migration to provide processes in a *Computing Community* with attributes such as collaborative work, distributed systems management, automatic reconfiguration, and fault tolerance [16]. In addition, we feel that general-purpose process migration affords significant far-reaching benefits, making it useful for much more than just load balancing:

- Consider a user actively using a complex application (such as a spreadsheet with lots of macros) that now wants to be able to work on another computer at another location without closing the application,

needing therefore to migrate the application. In another scenario, the user leaves the application running at work and then goes home and realizes she needs to do additional work. Process migration support would enable her to simply move the application over to the home machine. Further, the seamless "migratability" of applications between desktops and laptops can add another dimension to mobile computing.

- If we can migrate the application and its screen and its active connections to networks and files, then using the same mechanism, we should be able to move the screen without moving the application process. Decoupling the various external interfaces of a process (GUI, network connections, files, etc.) from the internal state of the process facilitates many interesting collaborative work situations. Taking this one step further, decoupling the internal process state from its interactions with operating system components such as dynamically linked libraries (DLLs) permits on-the-fly replacement of DLLs.

- The essence of process migration is the ability to capture all of the state that describes process behavior. Given such capabilities, application functionality can be extended using novel abstractions such as "active versioning". For example, while a user is working on developing complex macros for a spreadsheet, he might decide to do some risky experimentation without necessarily saving the current state (not just the files but the entire environment). It is only later that the user decides whether to commit these changes or revert back to an earlier state.

- Consider a server running a set of objects being actively used by a lot of external clients over the Internet. The system administrator needs to shut this machine off for maintenance but does not want to disrupt the service. The server along with its state and even active network connections can be migrated to another machine.

This focus on the novel capabilities provided by process migration mechanisms motivates a different approach. We aimed to be able to cleanly migrate any application process (without leaving any trace behind), irrespective of the behavior of the process. We do not want access to source code of the application, to modify the application or to re-link the application. While achieving this *might* seem very difficult, we next describe our approach that meets these requirements.

3. ARCHITECTURE

The process migration facility relies on a set of mechanisms to provide its services. It consists of a DLL that we call the PM-DLL (or Process Migration DLL). For all processes that are to be migratable, we inject the PM-DLL into the process and create a control thread in the process. This thread then reroutes all API accesses made by other threads in the process, by changing the entries in the DLL import address table. Hence, all relevant Win32 calls are now routed thought stubs in the PM-DLL.

The virtual environment is setup by the loader process (figure 1). The loader has two main responsibilities: to create a new process in a suspended state and to inject into it the DLL that will perform the API Interception. In the case that the application is being restarted on a new machine, the loader process would need to detect this, as it is partly responsible for restoring the process's state.

This responsibility is also shared by the PM-DLL's control thread. Additionally, the PM-DLL contains logic to checkpoint the process state just before migration. The PM-DLL is thus multi-threaded in order that it can perform all its duties simultaneously.

The loader also serves one more purpose; to notify the PM-DLL to begin check-pointing the application process state when the user requests that the application be migrated to a new location.

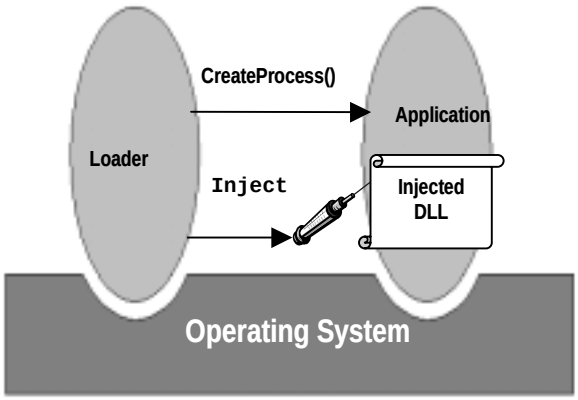

Figure 1: Process creation and DLL injection by Loader process.

4. IMPLEMENTING THE MIGRATION

A user on a computer participating in a *CC* can request that a local process of an application created by the loader be migrated to a machine of her choice. In order to make this happen, the loader and the PM-DLL save relevant state information pertaining to the process into binary files on some globally accessible storage. Thereafter the process ceases to exist on that machine. On the target machine, chosen by the user to run the migrated process, the loader is utilized to create a new process of the same application. While doing so, the loader becomes aware that the newly created suspended process is the result of a

migration and that there exists state information on files somewhere that must be used to bring this process to its last running state. As a result, the loader begins restoring the state information from the files. The PM-DLL completes this restoration and now the new process appears to the user in the same form as the earlier process, giving the perception of migration.

Following are the components of the process that together form its state.

Memory and Resources
Threads
Network connections
Graphical User Interface (GUI)

Figure 2: Components of a Process' State

We describe the method to save and restore each of these components. Collectively, this logic would enable the virtualized process to be migrated within a *CC*.

4.1 MEMORY AND RESOURCES

A running process affects the contents of its heaps and data section. In order to provide the process created on the new machine the same state as the recently migrated process, these heaps and data sections have to be recreated with the exact same data as in the earlier process. In a similar way, resources, like files, would have to be made available for the new process to use.

Data Section: The data section of a process consists of many regions. All except the *.data* region contain global data and constants and are therefore of no significance to process migration. These regions are automatically created from the executable when the process is started.

At the point of migration for a process, the control thread that exists as part of the injected DLL code identifies the *.data* region and saves its contents into a binary file on some globally accessible storage. To discover the location and size of this region, the control thread uses relevant headers from the Portable Executable File Format [27].

As part of state information that is written onto the newly created suspended process on the target machine, the injected control thread retrieves the binary information contained in the designated file and writes its contents into the memory region identified for *.data*.

Heaps: Information about the heaps is obtained from the process environment block (PEB). The PEB contains information about the number of heaps in the process and their addresses. Further, each *VirtualAlloc()* call is intercepted and information about the allocated memory is stored in data structures maintained by the injected DLL.
At the point of migration, the heap addresses and their allocated regions of memory can be saved into globally accessible binary files. Later, restoration will cause them to be created at exactly at the same locations in the process's virtual address space on the target machine.

Resources: 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 3). 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. Therefore even if we do not get the same object handle after migration, this handle translation performed by the vOS enables the process to continue as though there was no change to the resources it holds. Handle translation is implemented in the form of a Handle Translation Table that is maintained by the injected DLL code.

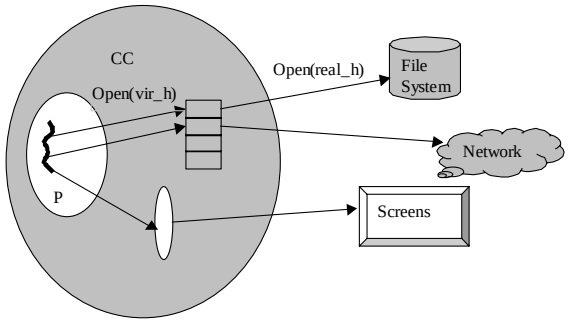

Figure 3: A process (P) in virtual execution environment in the CC and Translation of virtual handles to actual physical handles

4.2 THREADS

A thread is the basic unit scheduled by the operating system. Every process starts from a single primary thread and can create additional threads from any of its threads. From a process migration view, all the currently existing threads have to be restored with the same state on the destination machine.

To discover the threads of a process, we intercept the *CreateThread()* function and save the parameters that were used in the call into a data structure called the Thread Table. A thread can be in any one of the following states at the point of migration: running, suspended, wait and terminated. Depending on the state of the thread at the point of migration, the PM-DLL's control thread treats it differently.

4.2.1 THREAD CONTEXT AND STACK

At the point of migration, all process threads, irrespective of state, are suspended. The context of each thread is then obtained using the *GetThreadContext()* system call and saved into a binary file on globally accessible storage. At the destination machine, the loader creates all threads in a suspended state exactly how they were on the source machine. After the restoration of the stack and heaps, each context file is read to retrieve the context information into a data structure. This is then given to the *SetThreadContext()* API to restore the context for each thread.

Saving and restoring the stack involves a little more than a system call. When a thread is created, the system reserves a region in the address space for the stack and makes the second page of that region, the guard page, by using the *PAGE_GUARD* flag. This page acts as a boundary for the stack and allows the stack storage to increase only when the thread requires it. The flag serves as a good indicator of the stack. By using the *VirtualQuery()* function, we query the whole usable memory area and locate a region whose protection attributes indicate the existence of a guard page. Thus we have located the stack. The query function also gives us the base address, and size of the pages we query. This information can be stored in a binary file on globally accessible storage, along with the stack data.

Soon after it creates the new process in a suspended state, the loader will restore the stack and heap information using the *WriteProcessMemory()* system call. After this, it employs the *CreateRemoteThread()* system call and the information from the Thread Table to recreate all threads in suspended state.

4.2.2 THREAD STATES

Threads in Running State: For threads in the running state, restoration is very simple. At the destination machine, the loader creates all threads in a suspended state with the parameters that were recorded in the Thread Table. Each thread's context is set using the *SetThreadContext()* system call. Consequently, the operating system's scheduler determines the next thread to schedule.

Threads in Suspended State: In the case of threads that are to be recreated in a suspended state, the control thread at the destination would need to pass up resuming the thread after its context has been set. We set a field in the Thread Table to indicate whether the control thread should let this thread remain in a suspended state or resume it for operating system scheduling. Our discovery of the suspended state of a thread is due to the interception of the *SuspendThread()* and *ResumeThread()* system calls by the PM-DLL.

Threads in Wait State: The migration of a thread that is in the wait state is not as trivial. Apart from recognizing and remembering the state of the thread, we also need to record the objects that the thread is waiting for. The recreation of this at the destination machine is complicated by the following two factors:

1. Thread state is maintained inside the kernel and is inaccessible due to the lack of a programmer's interface. So we cannot create a thread in a wait state.

2. The loader process recreates threads, and all the objects are recreated by the injected PM-DLL's control thread. Both the loader and the PM-DLL exist in separate process address spaces. Therefore neither of them can make any created thread wait on particular objects.

A thread might be in a wait state because it has called a wait function like *WaitForSingleObject()*. Here we use the example of a thread that has called the *WaitForSingleObject()* function to explain how we implement the migration of threads in this state. On calling this function, the thread will enter into a wait state if the object is not signaled. The instruction pointer points to the line following the function call so that when this function returns, the calling thread runs from the code pointed to by the instruction pointer. If the user requests migration when the calling thread is waiting and before the object has been signaled, we cause the function to return with a hard-coded return value. This is possible because the PM-DLL has intercepted this function call and redirected the call to a wrapper function. It is the wrapper function that in reality returns with the hard-coded return value. This value is stored in a variable and is not the same as any valid return value that the function could have had. At the destination machine, during restoration, the PM-DLL's control thread executes a conditional statement that checks the value of this variable and determines if the wait function was forced to return due to a migration request. The PM-DLL knows to perform this check because the wait functions have been intercepted and logged when they were called at the source machine. In the case that the variable holding the return value of the function has the specific hard-coded value, we cause the wait function to be executed again. Therefore, the thread continues to wait on the same objects.

Threads in Terminated State: Threads can reach the terminated state in one of the following three ways: by calling *ExitThread()* or *TerminateThread()*, or, by a *return* statement at the end of the thread function. The PM-DLL intercepts the *ExitThread()* and *TerminateThread()* call and deletes the thread handle from the Thread Table if the call is made. Thereafter the thread record is not available and the thread is not recreated.

Records of threads that terminate due to a *return* statement are not removed from the Thread Table until the user requests migration. At this point, the control thread tries to suspend all threads in order to save relevant information for recreation. If the *SuspendThread()* call fails, the control thread assumes that the thread had reached a terminated state. The record of the thread handle is then removed from the Thread Table.

Synchronization objects: For the purpose of synchronization, threads sometimes create kernel objects like, events, timers, semaphores and mutexes. The states of these kernel objects will cause the thread to transition its own state. Therefore keeping track of the state of these kernel objects would help us recreate all threads in their original state after the migration.

Intercepting the system calls that create these kernel objects or change their state gives the PM-DLL information that can be used during restoration.

4.3 NETWORK CONNECTIONS

This paper deals only with migrating processes created by a loader within a *CC* that may have active network connections with another process also created by a loader within the same *CC*.

At process creation, the injected DLL intercepts all the important socket APIs and wraps them with wrapper APIs. Further, it creates a socket thread for the purpose of closing and restoring network connections when the process migrates to a new machine.

Migrating a client process: Migrating a client process is a bit different from migrating a server process. When a client has to migrate, the control thread saves information about all the sockets into a binary file on some globally accessible storage. The control thread then sends the port number of the client socket and a request to close the connection to the socket thread on the server. When it receives an acknowledgement from the socket thread on the server, it closes the connection and reports this to the server. Now, the socket thread on the server also closes its connection and waits for a message from the migrated client process.

After the client process is restarted on a new machine, the control thread restores the process state and reads the socket information from the binary file. It also retrieves the server IP address and port number from the file. The control thread then sends a message to the socket thread on the server that it is ready to reopen the connections. As a result the server's socket thread creates a socket that binds to the same port as it was before and waits for the connection. The client's control thread then makes a connection to the server and saves the new socket handle. However, due to virtualization, the application still sees the same virtual socket handle.

Migrating 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 new machine, it sends the IP address of the new machine to the client's socket thread. This allows the client to know the server's new location and re-establish all the connections.

4.4 GRAPHICAL USER INTERFACE

The window object is inserted into the Handle Table of the injected DLL by intercepting the *CreateWindow()* and *CreateWindowEx()* calls. In addition, the injected DLL maintains a history of messages that were sent to the window object by intercepting *SendMessage()* and *DispatchMessage()*.

During restoration, the control thread is charged with re-creating all the objects that are present within the Handle Table. The window object is thus created to look exactly like the original process window. The window's message pump checks the history of messages and sends them again to the new window.

5. PERFORMANCE

The performance tests were run on Intel Pentium machines with 1 GHz CPU frequency and 512 MB RAM. In our tests we measured the following parameters:

1. Time required for loading the process without any virtual environment or PM-DLL.
2. Time required for loading the PM-DLL and creating the virtual environment.
3. Time required to checkpoint and restore state information.
4. Space requirements in order to store the state information between the checkpoint and restore phase.

We used 2 processes for the measurements. Process A ran a dummy Win32 program consisting of 2 threads, 10 heaps, 15 objects and invoked some file I/O API's. Process B ran the unmodified Wordpad executable available with the Windows operating system.

Process Creation: The first measurement is the time taken by Windows 2000 to create the processes. Process A takes 1.49 milliseconds to be created and Process B takes 1.53 milliseconds (the difference in negligible).

Loading the PM-DLL: The number of API calls that need to be intercepted by wrapper functions varies in each

application. The size of the PM-DLL is directly proportional to this number. The PM-DLL for Process A had wrappers for 24 API calls, while the PM-DLL for Process B had wrappers for 340 API calls.

The time taken to install the wrappers for these processes (after the process has been started) was 33 milliseconds for Process A and 259 milliseconds for Process B. The difference shows the extra work (and overhead) of wrapping a larger number of API calls.

The difference between the first measure and the second one gives us the overhead of our architecture in terms of extra time taken at process startup.

Checkpoint and Restore: Process migration is done in two separate and independent steps – checkpoint and restore. In addition, while the process executes, the virtual environment is constantly monitoring object handles and translating them using a Handle Table. The creation of an object handle needs 6.6 microseconds, while the time required for deletion and matching operations varies according to the size of the Handle Table.

When a migration event happens, the checkpoint time depends on the number of threads, heaps, objects. For Process A, the checkpoint time was 88.6 milliseconds, and the checkpoint information size was 1.15MB. On the destination machine, the restoration for Process A took 68.6 milliseconds.

For Process B the checkpoint took 1088 milliseconds and the restore took 372 milliseconds and the checkpoint information took 1.76MB. The very high numbers on this process is due to a later discovered inefficiency in the manner heap data is being recorded, and we are working on an optimization.

6. RELATED WORK

MPVM (Migratable PVM): MPVM [22] is an extension of PVM that allows parts of a parallel computation to be suspended and resumed later on other workstations. Transparency is ensured by modifying the PVM libraries and daemons and by providing wrapper functions to certain system calls so that migration occurs without modifying the application code.

Condor: This [17] is a distributed batch processing system for UNIX that can transparently checkpoint the process state to a file and restart the process on a different machine. Programs are re-linked to include the checkpoint libraries. It is implemented at the user level.

Libckpt: This [28] is a portable transparent checkpointing library on UNIX. It checkpoints the process state using transparent incremental and copy-on-write checkpointing. To use libckpt, the developer has to change a line of his source code and recompile with the libckpt library. It runs at the user level.

Sprite: This [26] 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.

A Transparent Checkpoint Facility On NT: This [29] implements a checkpoint facility on NT, 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.

NT-Swift (Software Implemented Fault Tolerance On Windows NT): NT-SwiFT [30] 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.

7. CONCLUSION

We present results in providing general-purpose processes with a migration facility. Migrating processes has many advantages, including the ability to make them mobile and the ability to make them tolerant to failures. While previous work has focused on scheduling and checkpointing special processes, we do the same for general processes. In addition this facility does not need any libraries to be linked, access to source code, or modifications to the program

8. ACKNOWLEDGEMENT

This research is partially supported by grants from DARPA/Rome Labs (F30602-99-1-0517), NSF (CCR-9988204)-Intel and Microsoft, and is part of the "*Computing Communities*" project, a joint effort between Arizona State University and New York University.

REFERENCES:

[1] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck and V. Sunderam, PVM: Parallel Virtual Machine, *The MIT Press*, 1994.

[2] W. Gropp, E. Lusk & A. Skjellum, Using MPI Portable Parallel Programming with the Message Passing Interface, *MIT Press*, 1994, ISBN 0-262- 57104-8.
[3] P. Dasgupta, Z. M. Kedem & M. O. Rabin, Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach, *Proc. of the 15th IEEE International Conf. on Distributed Computing Systems*, 1995.
[4] N. Carriero & D.Gelernter, Linda in Context, *Comm. of ACM*, 32, 1989.
[5] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, & W. Zwaenepoel, TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, December 1995.
[6] E. Speight & J. K. Bennett, Brazos: A Third Generation DSM System, *The USENIX Windows NT Workshop*, 1997.
[7] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen & G. van Rossum, Experiences with the Amoeba Distributed Operating System, *Communications of the ACM, 33(12)*, 1990.
[8] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M.B. Jones, Mach: A System Software Kernel, *Proc. of the 1989 IEEE International Conf. COMPCON*.
[9] P. Dasgupta, LeBlanc Jr., R. J., M. Ahamad, & Ramachandran, The Clouds Distributed Operating System, *IEEE Computer*, Nov. 1991.
[10] M. Rozier, V. Abrossimov, F. Arm & M. Gien, M. Guillemont, F. Hermann & C. Kaiser, Chorus (Overview of the Chorus Distributed Operating System), *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architecture*s, 1992.
[11] T.Boyd & P.Dasgupta, Injecting Distributed Capabilities into Legacy Applications Through Cloning and Virtualization, *International Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, July 2000.
[12] R. Nasika & P. Dasgupta, Transparent Migration of Distributed Communicating Processes. *13th ISCA International Conf. on Parallel and Distributed Computing Systems (PDCS-2000),* August 2000.
[13] P. Dasgupta, V. Karamcheti and Z. Kedem, Transparent Distribution Middleware for General Purpose Computations, *International Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June 1999.
[14] S. Sardesai & P. Dasgupta, Chime: A Windows NT based parallel processing system, *USENIX Windows NT Symposiu*m, Seattle, August 1998. (Extended Abstract).
[15] J. Richter, *Applications for Windows* (Microsoft Press, 1997).
[16] D. McLaughlin, S. Sardesai, & P. Dasgupta, Preemptive Scheduling for Distributed Systems, *11th International Conf. on Parallel and Distributed Computing System*s, 1998.
[17] M. Litzkow, M. Livny & M. Mutka, Condor—A Hunter of Idle Workstations, *8th International Conf. on Distributed Computing System*s, 1988.
[18] E. Jul, H. Levy, N. Hutchinson, & A. Black, Fine-grained Mobility in the Emerald System, *ACM Transactions on Computer System*s, *6(1)*, 1988.
[19] A. Baratloo, M. Karaul, Z. Kedem, & P. Wyckoff, Charlotte: Metacomputing P. Wyckoff. Charlotte: Metacomputing on the Web, *FutureGeneration Computer System*s, 1999.
[20] S. Sardesai, D. McLaughlin & P. Dasgupta, Distributed Cactus Stacks: Runtime Stack-Sharing Support for Distributed Parallel Programs, *International Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98*), July 1998.
[21] V. S. Sunderam, PVM: A Framework for Parallel Distributed Computing, *Concurrency—Practice and Experienc*e, *2(4)*, December 1990.
[22] J. Casas, D. L. Clark, R. Konoru, S. W. Otto, R. M. Prouty & J. Walpole, MPVM: A Migration Transparent Version of PVM, *Computing Systems: The Journal of the USENIX Association, 8(2)*, Spring 1995.
[23] L. Dikken, F. van der Linden, J. J. J. Vesseur, & P. M. A. Sloot, DynamicPVM: Dynamic Load Balancing on Parallel Systems, In W. Gentzsch & U. Harms, editors, *High Performance Computing and Networkin*g, Springer Verlag, LNCS 797, April 1994, 273-277.
[24] A. Barak, S. Guday & R. G. Wheeler, The MOSIX Distributed Operating System, *Lecture Notes in Computer Scienc*e, *Vol. 672*, Springer, 1993.
[25] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers & C. Chambers, Extensibility, Safety and Performance in the SPIN Operating System, *15th Symp. On Operating Systems Principle*s, December 1995.
[26] F. Douglas & J. Outerhout, Process Migration in the Sprite Operating System, *Proc. of the 7th International Conf. on Distributed Computing System*s, September 1987, 18-25.
[27] M. Pietrek, Peering Inside the PE: A tour of the Win32 Portable Executable File Format. *Microsoft MSDN Library*, March 1994.
[28] J. S. Plank, M. Beck & G. Kingsley, Libckpt: Transparent Checkpointing under UNIX, *Proc. USENIX* Winter 1995, New Orleans, Lousiana, January 1995.
[29] J. Srouji, P. Schuster, M. Bach & Y. Kuzmin, A Transparent Checkpoint Facility On NT, *Proc. of 2nd USENIX Windows NT Symposiu*m, August 3-4 1998.
[30] Y. Huang, P. E. Chung, C. Kintala, D. Liang & C. Wang, NT-SwiFT: Software Implemented Fault Tolerance for Windows NT, *2 nd USENIX Windows NT Symposiu*m, July 1998.