Design and Implementation of a CPU Bound Process Migration in Windows 7

AbdulSattar M. Khidhir
Mosul Technical Institute
Foundation of Technical Education
Mosul, Iraq
abdulsattarmk2@yahoo.com

Bassam A. Mustafa, Nadia T. Saleh
Dept. of Computer Science
University of Mosul
Mosul, Iraq
bassamali2004@yahoo.com, nadia_trk_s@yahoo.com

***Abstract*—This paper presents a new mechanism for user-level process migration in a network of homogeneous systems running Windows 7 operating system. The methodology supports the migration of execution entities among nodes during runtime. Within this approach the execution and memory states of a process are transferred dynamically from one node to another in the distributed system. Some other techniques are essentially required in order to achieve migration. These techniques are suspending, checkpointing, and resuming the transferred process.**

Keywords: process migration; process checkpointing and restart; CPU bound.

I. INTRODUCTION

A binary program image is a passive entity that describes a computation, which is executable by a computer, such as the contents of a file stored on disk, whereas a process is an active (dynamic) entity of this binary program image that is actually under execution in the underlying operating system [1], [2].

The process's state, during the process execution, consists of two interconnected states: a static state and a dynamic state [1].

Static state is characterized by the binary program image, i.e., the executable program file. Whereas the dynamic elements like: the process credentials, CPU-registers, stack, virtual memory, physical memory, occupied and requested I/O devices, system call under execution, regular-files opened, signals received, sockets established, interprocess communications, currently working threads, and other terms characterize the dynamic process state [1].

The Process migration is the act of transferring the dynamic (active) process state between two machines and restoring the process from the point it left off on the selected destination machine [3]. The reasons behind this are to provide an enhanced degree of dynamic load distribution, fault tolerance, effortless system administration, resource sharing, data access locality, mobile computing, collaborative work, distributed systems management, and automatic reconfiguration. Furthermore, all these can be achieved without modifications to the kernel or the applications, consequently the term 'process migration' has divers meanings in modern computing. In a large network of workstations and with the ever-increasing role of the World Wide Web, the potential of process migration is significantly obvious [3], [4].

In spite of the high cost and intricate tasks involved in moving a running process to a different machine, migration grants system performance, which represents the major goal. System performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines, i.e., load balancing, which means explicit positioning of processes to distribute the computational load [3], [5], [6], [7].

A process load is often characterized by the CPU queue length or CPU utilization, memory usage, communication, files usage, etc [3], [5].

Another benefit of migration is resource sharing, where nodes having large amount of resources can work as receiver nodes in a process migration environment. Running processes on a specified node that might be distant from other nodes, which house the data that the processes are using, tend to spend most of their times in performing communication between the nodes for the sake of accessing the required data. Process migration can be employed for transferring a distant process closer to the processed data, thereby ensuring it spends most of its time doing useful work [3].

Higher degree of parallelism is also achieved by mapping processes to machines at runtime [5].

One of the most important aspects of any system is fault tolerance, which can be improved by migration of a process from a partially failed node, or in the case of long running processes when various kinds of failures are possible. The system can migrate processes, which are running on one node to another node when the source node is about to be shutdown. Also, the system could enable processes to go to completion on the new node [3]. On the other side, checkpointing process could be used for recovery in case of system crash [7].

Long running applications that can run for days or weeks on one node can undergo different types of interruptions. Migration system can relocate these processes in case of the occurrences of any of the events mentioned previously [3].

In addition to the performance improvement, there are other motivations for supporting code migration as well. The most essential one is flexibility. This could be noticed in building distributed applications by partitioning the application into different parts and deciding in advance where each part should be executed [5].

978-0-7695-4959-0/13 $25.00 © 2013 IEEE
DOI 10.1109/ACSAT.2012.30

This paper gives a proposed system implementation for process migration in windows 7 environment using Visual C++ programming language, as both of them don't support this mechanism.

The remainder of the paper is structured as follows:

Section 2 gives an overview about the related works. Section 3 involves the models of migration. In section 4 the main process migration activities and related environment are discussed. The design issues of migration system are presented in section 5. In section 6 the details of design implementation are explained. Finally, section 7 concludes the paper.

II. Related Works

Most of the existing works related to process migration is performed on Linux operating system and few of them are done on Windows NT. To our knowledge, there is no such work on Windows 7. The following works are chosen to be discussed as related works.

Checkpoint facility on NT [8], which has been developed at Intel Corporation, describes the implementation of a checkpoint library that permits users to save temporary state of long-running multithreaded programs on a Windows NT system and to resume execution from the checkpointed state at a later time. 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.

User-Level Thread Migration and Checkpointing on Windows NT Clusters [9] explains the design and implementation of two user-level mechanisms (thread migration and checkpointing) in the Brazos parallel programming environment that address these issues on clusters of multiprocessors running Windows NT. Brazos is a distributed system that supports both shared memory and message passing parallel programming paradigms on networks running Windows NT operating system with Intel x86-based multiprocessors architecture.

Transparent Migration of Distributed Communicating Processes [10], a joint effort between Arizona State University and New York University, describes the concept of virtualization and the mechanism of injection and the implementation of a wrapper DLL on Windows NT. It shows how processes can migrate between machines without disrupting the socket communications. Process migration mechanism is implemented here without any modification to the application or OS, by using the Win32 API interception mechanism.

Software Implemented Fault Tolerance on Windows NT (NT-SwiFT) [11] is a set of components that facilitates building fault tolerant and highly available applications on Windows NT. It has the ability to checkpoint data segment, communication channels, threads' contexts, stacks, etc. NT-SwiFT provides detection of failure and error recoveries for both client-side and server-side programs.

SNOW project [12] implements a methodology to support user-level process migration for traditional stack-based languages such as C and FORTRAN in heterogeneous distributed environments. This methodology addresses the three outstanding problems of transferring execution state, memory state, and communication state.

Process Migration in Network of Linux Systems [13] deals with the design and implementation of process migration in network of homogeneous systems running on Linux. Also, the Checkpoint/Restart System (CRS) has been developed on top of the migration system to run as an application. The software product doesn't require modification of existing kernel, but part of the CRS is implemented as a kernel module to achieve better transparency and performance. The software can migrate only well defined, CPU bound processes. It cannot migrate these processes that have child process, open files, signals, and socket communication.

In [14] Joshi and Choksi discuss a kernel-level solution for the problem of checkpointing the desired virtual memory areas of a process. Also, in [1] they discuss another kernel-level solution for the problem of checkpointing the credential of a process identifier (PID) and allocating a selected process-id value to a new process. Both of these papers suggest solutions for the Linux kernel 2.6.25 environment.

Another work of Joshi and Choksi [15] describes load balancing techniques to share the workload of the workstations that belong to a particular network in order to obtain better performance from the overall network. This paper presents the mechanism of load information collection, how to determine the idle workstations as well as highly-loaded workstations, and how to distribute the load from one workstation to another. Moreover, a dynamic load balancing algorithm, which achieves runtime performance gained by managing the jobs as they arrive, is discussed here.

Load Balancing by Process Migration in Distributed Operating System [16] focuses on process migration technique for load balancing. It describes two algorithms: sender-initiated algorithm and receiver-initiated algorithm. In sender-initiated algorithm, non-preemptive migration is implemented because sender (overloaded node) usually wants to send its newly arrived process to another node rather than executing it. While in receiver-initiated algorithm, lightly loaded node wants to receive a process from the overloaded node, consequently preemptive migration is implemented with this algorithm. The work is implemented in Linux operating system at kernel-level.

III. Models for Code Migration

Conventionally, distributed systems communication is concerned with exchanging data between processes. Code migration is the term that depicts moving programs between machines, with the intention to have those programs to be executed at the target [5].

A process consists of three segments, namely code segment, resource segment, and execution segment. The first segment contains the set of instructions that make up the program that is being executed. The resource segment consists of references to external resources that might be used by the process, such as files, printers, devices, other processes, etc. Finally, the execution segment stores the current execution state of a process, consisting of private data, the stack, the program counter, and other registers [5].

The basic model for code migration should provide, at least, a weak mobility. In this model, only code segment is transferred, along with perhaps some initialization data. The significant feature of weak mobility is that the transferred program is always started from its initial state, which makes this model a simple approach [5].

Another model that is in contrast to weak mobility is strong mobility. In the systems that support this model the execution segment can be transferred as well. The most important feature of strong mobility is that a running process can be suspended, moved to another machine, and then resume execution where it left off [5]. Obviously, strong mobility is much more powerful than weak mobility, but also much harder to implement and more expensive as the collection of a process's state, which could be quite large and complex, can be difficult [5], [16].

Another classification of process migration types places them into two classes: Preemptive process migration and non-preemptive process migration. Preemptive type, sometimes called dynamic, involves transferring a process that is partially executed, which resembles the strong mobility type [14], [16].

On the other hand, non-preemptive process migration, which is also called static, involves the transfer of processes that have not begun execution and hence do not require the transfer of the process's state. This type represents the weak mobility [14], [16].

Dynamic or preemptive process migration minimally requires suspending the execution of a process on the source node, extracting the process's state, relocating the extracted state to the destination node, reconstructing the state on the destination node and then resuming the process's execution on the new node [14].

IV. Migration Process

Although transferring and reconstructing the process's state are considered as important activities, the major activity involved in process migration is checkpointing [13].

Reconstructing implies restoring the checkpointed state of the process on the destination node and then resuming the execution of that process based on the saved state, from exactly the point of suspension [13].

Checkpointing implies suspending and saving the process's state on the source node [13].

Besides these activities, the related environment plays an important role in migration process. The following subsections illustrate some aspects that belong to these activities.

A. Checkpointing

Checkpointing a specified process is saving its state. As mentioned above, process's state includes register set, memory address space, allocated resources, and process private data [13].

Execution of the process can be later resumed from a checkpoint file. This would prevent losing data generated by long-running processes caused by system or program failures. Also, it would facilitate debugging, as bugs might appear after the program has executed for a long time [8].

Checkpointing can be implemented at two levels namely kernel-level and user-level [17].

In the first type, the operating system supports checkpointing and restarting processes. Kernel-level checkpointing is transparent to applications. Besides, the applications do not have to be modified or relinked with any specified library to support checkpointing mechanism [13].

This technique modifies the operating system kernel to make process migration more efficient. These modifications allow the migration procedure to be done faster. Additionally, more types of processes could be migrated. Unfortunately, many kernel-level implementations have high overhead, long checkpointing time, and still cannot migrate all processes [17].

User-level checkpointing allows application programs to be checkpointed and linked with a checkpoint user library. Before starting this mechanism, a checkpoint-triggering signal is sent to the process. The user library's functions respond to the signal and save the information necessary for restarting the process again. After that and at restart, the functions in the checkpoint library restore the saved state for the process [13].

This technique doesn't change the operating system kernel. It is easier to be developed and maintained but has some common problems. They cannot migrate all processes because they cannot access kernel state. Also, they must use kernel requests to cross the kernel/application boundary, which are slow and costly [17].

B. Migration Environment

Process migration is performed under one of the following environment types [17]:

Homogeneous process migration environment; where all systems have the same architecture and operating system but not necessarily the same resources or capabilities [17].

Although homogeneous type offers good use of available network resources, it is only applicable between machines of same architectures and operating systems. Networks may contain different types of machines running different operating systems with resources available on machines of different architectures than the current machine where the process is executing. Such diversity leads to heterogeneous process migration environment [17].

Heterogeneous process migration is performed across machine architectures and operating systems. Clearly, it is more sophisticated than the homogeneous type because it must consider machine and operating specific structures and characteristics, in addition to its ability to transmit the same information as homogeneous type [17].

Mobile environment is the more appropriate environment for using heterogeneous process migration. Because within this environment it is more likely that the mobile unit and the base support station will be of different machine types. A process could be migrated from the mobile unit to the base station and vice versa during computation. In most cases, this could not be achieved by homogeneous migration [17].

In spite of these advantages, efficient process migration in a heterogeneous environment faces a great challenge [12].

In such environment, the execution state of a process cannot simply be transferred by copying context and register sets across machines. There is an essential need for machine-independent mechanisms for transferring the execution state. Problems arise in transferring the memory state due to differences in memory configuration and memory management among heterogeneous computers. The data that is stored in memory storages of different machines could be represented in different formats, different memory addressing scheme, etc. Consequently, there must be a mechanism to capture information in a process memory space as well as to migrate them to a new machine [12].

V. DESIGN

The design of process migration system consists of two main subsystems:

- The source node subsystem, which performs the checkpointing mechanism and transfers the checkpointed information for the selected process.
- The destination node subsystem that receives the previously mentioned information and restarts the process again.

A. Suspending and Marshalling Algorithm

The following steps give the outline activities performed at source node subsystem:

Step1: Select a process to be migrated. The process is a CPU bound process.

Step2: Suspend the selected process.

Step3: Read process state, which consists of thread context, thread stack, and memory address space of code and data. Store the information on specified files prepared for this purpose.

Step4: Resume or terminate the suspended process depending on an option.

Step5: Transfer the information files to the selected destination node.

B. Deployment and Resuming Algorithm

The following steps describe construction and restarting mechanisms at the destination node:

Step1: Receive the checkpointed information (files) from the source node.

Step2: Create a new process.

Step3: Read the thread context of the new process and replace it with the checkpointed thread context that is stored in a file.

Step4: Copy the contents of code, data, and stack sections from the specified files to the new process address space.

Step5: Resume the new process execution.

Both subsystems' algorithms are depicted in Fig. 1 and Fig. 2 below.

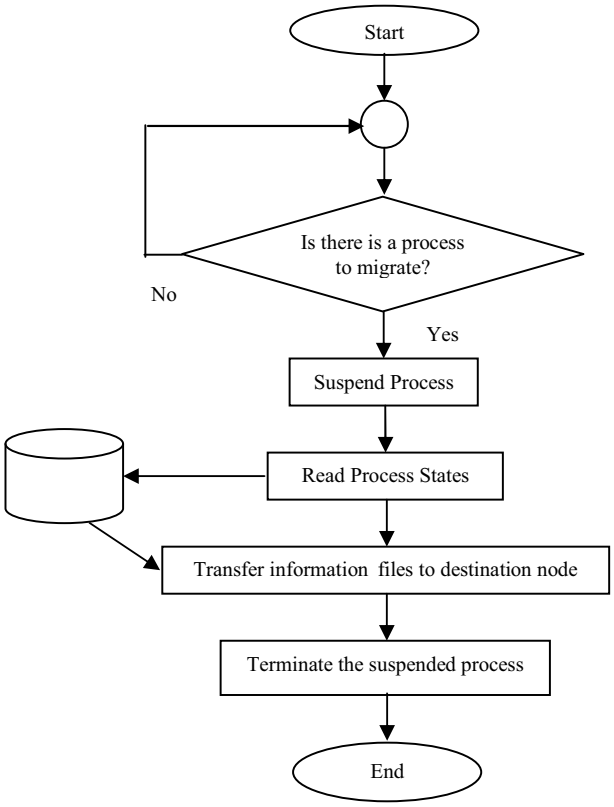

Figure 1. Suspending and Marshaling Process.

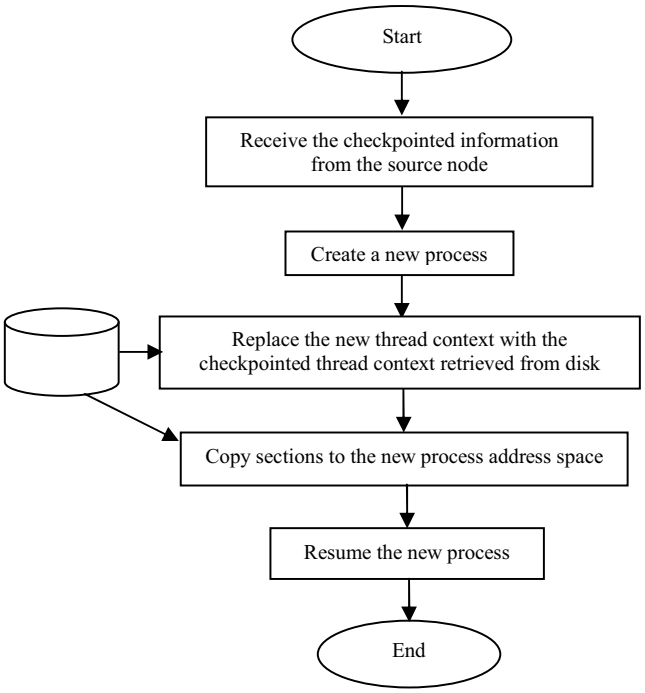

Figure 2. Deployment and Resuming Process.

VI. IMPLEMENTATION

The implementation of the two subsystems is performed using Win32 API functions in the Microsoft Visual C++ 2010 Ultimate environment, working on windows 7 operating system.

As mentioned above, the work is divided into two subsystems. The next subsections give the details for implementation mechanism.

A. Source Node Implementation Issues

The subsystem at the source node selects a process for migration. The selected process should satisfy some conditions to be able to migrate. The current conditions assume that the process is a CPU bound process.

After CreateProcess win32 API function is called to launch the process, another API function is called to suspend the process's associated thread, it is SuspendThread function. At this point GetThreadContext is called to retrieve the context of the thread. The thread's context consists of all the register values like program counter, stack pointer, general purpose registers, etc. The context is saved in a specified file to be transferred later to the destination node.

By default, the virtual size of a process on 32-bit Windows is 2 GB. The virtual address space is divided into units called pages. Any page in a process virtual address space could be either free, reserved, or committed. Regardless of the amount of physical memory that is actually available, to the application it seemed that there is 2 GB of memory available [18], [19]. The application can first reserve address space and then commit pages in that address space, or it can reserve and commit pages in the same function call [18].

Reserved address space is simply a way for reserving a range of free virtual addresses for future use, protecting the addresses from other allocation requests [19]. Attempting to access an address that is either reserved or free results in an access violation exception. This is because the page isn't mapped to any storage that can resolve the reference. Committed pages are translated to valid pages in physical memory when they are accessed [18].

The Win32 API VirtualQueryEx function retrieves information about a range of pages within the virtual address space of the process. This function is called after getting the thread context. It needs to walk the process's address space identifying each of its distinct address regions and representing specific state information about each region. The function enumerates each region one at a time from the bottom of the process address space to the top [19]. A pointer to a MEMORY_BASIC_INFORMATION structure is passed to be filled in by the VirtualQueryEx function. This structure represents information about the region queried. Such information represents the lower bound of the region and the size of the region, along with the exact state of the addresses in the region [18].

Each thread has a user-mode stack that is reserved at the thread's creation (1 MB is the default size). By default, the initial page in the stack is committed and the next page is marked as a guard page. The guard page, which isn't committed, traps references beyond the end of the committed portion of the stack and expands it [18].

In order to find the thread's stack, first the thread's current stack pointer (ESP) is acquired, which is part of a thread's context. The Win32 VirtualQueryEx function is then used to determine the region of committed memory associated with the thread's stack.

Also, the starting address of the data region, which consists of global and static variables, can be found in the portable executable file format (PE) [20]. This address is mapped to the process's virtual address space and the size of the region is found by calling the same API function for that starting address.

The code region in the process's virtual address space is typically memory mapped from the executable stored on the disk. Information about this region is also gathered by calling VirtualQueryEx on the image base address taken from the PE file format.

After that ReadProcessMemory is called to read the data from the process's memory address space. These data are saved in a specified file to be transferred to the destination node.

B. Destination Node Implementation Issues

Upon receipt of the checkpointed information, the destination node subsystem creates a new process and suspends it with CreateProcess and SuspendThread functions, respectively. This subsystem sets the new thread's context using SetThreadContext. Some registers values should not be changed, instead, their values are incremented or decremented with specified offsets depending on the values of checkpointed registers. Such registers are the EIP, ESP, and EBP.

This subsystem also, copies the stack and code data with WriteProcessMemory function to the new process address space, and then activates its thread using the ResumeThread routine.

Fig. 3 and Fig. 4 show snapshots of the process's states for both source and destination nodes.

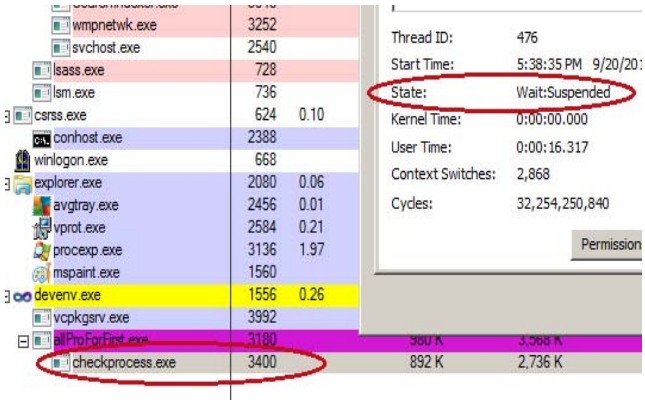

Figure 3. Source Node's Suspended Process .

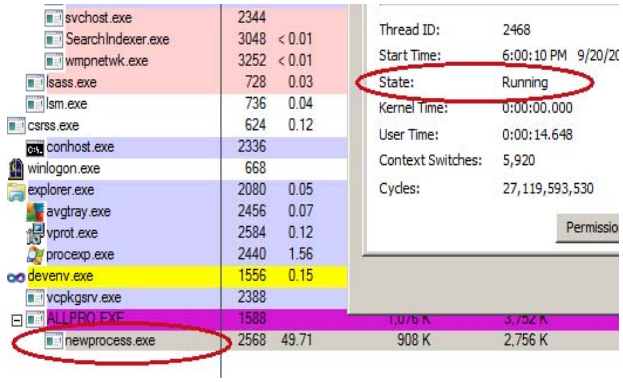

Figure 4. Destenation Node's Running Process.

VII. CONCLUSIONS

- The current design requires no modifications to the existing kernel, thus it is more suitable to work on Windows 7 because the source code of this operating system is not freely available. Besides, kernel programming is hard and error-prone, so it is better to perform functionality in user space rather than the kernel space.
- In the other hand, even if the design is implemented using Win32 API, it doesn't need to wrap any of the system API calls or intercept any API function.
- There is no need for both nodes (source and destination) to have the program code and stack loaded at the same virtual addresses. Destination node subsystem can adjust or remap any pointers (registers) by adding the appropriate offset rather than changing the actual value.
- C/C++ language has no features to support migration or heterogeneity. By investigating the requirements of such language, it may be possible to design a system that support this language and usable for today's widely distributed networks.
- Process can dynamically change its runtime environment by migrating to an upgraded or more appropriate computation platform. Also, resuming the migrated process from the checkpointed one could be done at any time and gives the same results as the original execution, which in turn leads to more flexible system and tolerate failures.
- Checkpoint procedure minimizes the costs of failures by preserving results of a run until the last good state before failure. It also provides an advanced state for debugging that permits programmers to correct their programs more easily.

REFERENCES

[1] N. A. Joshi and D. B. Choksi, "Forking a New Process with the Predetermined PID Value during Process Migration," in *Proc. of the 2nd National Conference on Information and Communication Technology (NCICT)*, 2011 by IJCA Journal, No.1 article 3, pp. 8-11.

[2] A. Silberschatz, P. B. Galvin, and G. Gang, *Operating System Concepts*, 7th ed., III River Street, Hoboken, NJ, USA: John Wiley & Sons, Inc., 2005.

[3] N. Vasudevan and P. Venkatesh, "Design and Implementation of a Process Migration System for the Linux Environment," in *Proc. of the 3rd International Conference on Neural, Parallel and Scientific Computations*, Morehouse College, IFNA, Atlanta, GA, USA, Aug. 9-12, 2006.

[4] M. Khambatti, "Checkpointing Process Heaps and Cleaner API Interception for Process Migration," MCS Project, Computer Science and Engineering Department, Arizona State University, Dec. 2000.

[5] A. S. Tanenbaum and M. Van Steen, *Distributed systems Principals and Paradigms*, Upper Saddler River, New Jersey, USA: Prentice Hall, 2002.

[6] P. Troger and A. Polze, "Object and Process Migration in .NET," in *Proc. of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems*, University of Potsdam, Germany, Jan. 15-17, 2003, pp. 139-146.

[7] K. Noguchi, "Spontaneous Process Migration with Global Pointers," Ph.D. Dissertation, University of California, Irvine, USA, March 2007.

[8] J. Srouji, P. Schuster, M. Bach, and Y. Kudmin, "A Transparent Checkpoint Facility on NT," in *Proc. of the 2nd USENIX Windows NT Symposium*, Seattle, Washington, USA, Aug. 3-4, 1998, pp. 77-85.

[9] H. Abdel-Shafi, E. Speight, and J. K. Bennett, "Efficient User-Level Thread Migration and Checkpointing on Windows NT Clusters," in *Proc. of the 3rd USENIX Windows NT Symposium*, Seattle, Washington, USA, July 12–13, 1999, pp. 1–10.

[10] R. Nasika and P. Dasgupta, "Transparent Migration of Distributed Communicating Processes," in *Proc. of the 13th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS-2000)*, Aug. 2000.

[11] Y. Huang, P. E. Chung, C. Kintala, C.-Y. Wang, and D.-R. Liang, "NT-SwiFT: Software Implemented Fault Tolerance on Windows NT," in *Proc. of the 2nd USENIX Windows NT Symposium*, Seattle, Washington, USA, Aug. 1998, pp. 47-55.

[12] K. Chanchio and X.-H. Sun, "SNOW: Software Systems for Process Migration in High-Performance, Heterogeneous Distributed Environments," in *Proc. of the International Conference on Parallel Processing Workshops*, 2002, pp. 589-596.

[13] Ch. D.V. Subba Rao, M.M. Naidu, K.V. Subbaiah, and N. R. Reddy, "Process Migration in Network of Linux Systems," *International Journal of Computer Science and Network Security (IJCSNS)*, vol.7, No.5, pp. 213-219, May 2007.

[14] N. A. Joshi and D. B. Choksi, "Checkpointing Process VMA for process migration," *Journal of Emerging Trends in Computing and Information Sciences*, vol. 2, Special Issue, pp. 7-10, Feb. 2011.

[15] N. A. Joshi and D. B. Choksi, "Mechanism for Implementation of Load Balancing using Process Migration," *International Journal of Computer Applications*, vol. 40, No. 9, pp. 16-18, Feb. 2012.

[16] V. Shah and V. Kapadia, "Load Balancing by Process Migration in Distributed Operating System," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 2, issue 1, March 2012.

[17] R. Lawrence, "A Survey of Process Migration Mechanisms," Dept. of Computer Science, University of Manitoba, Canada, May 29, 1998.

[18] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals: Covering Windows Server 2008 and Windows Vista*, 5th ed., Redmond, Washington, USA: Microsoft Press, June 17, 2009.

[19] R. Kath, "Managing Virtual Memory," a System Service Technical Article, Microsoft Developer Network Technology Group, Jan. 20, 1993.

[20] E. Eilam, *Reversing: Secrets of Reverse Engineering*, Indianapolis, Indiana, USA: Wiley Publishing, Inc., 2005.