Preemptive Module Replacement Using the virtualizing Operating System[1]

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

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

ABSTRACT
Software aging causes software programs to fail over time. Rejuvenation of the software is a preemptive methodology developed to reduce failure, which reduces the need for complex methods to identify and fix problems after a failure has occurred. It does not eliminate the need for managing failure, it simply moves the bulk of the processing to a more controllable and simpler pre-failure state.

We have developed the virtualizing Operating System (vOS) residing on top of Windows 2000. The vOS is a middleware application developed to create an abstraction layer between an application and the underlying operating system. The abstraction layer allows for the virtualization of resources that a COTS application uses. The approach to virtualizing the application is by injecting functionality into running applications. Using this scheme, we separate the physical resource bindings of the application and replace it with virtual binding, referred to as virtualization. We use this technique to migrate an active program between systems without the program's awareness or involvement. We describe how to extend this capability to provide for both preemptive module level refreshment and program restart without modifying the application or the operating system.

General-Terms
Management, Design, Reliability.

Keywords
Self Healing, Software Rejuvenation, Parallel/Distributed Computing Systems, Virtualization.

1. INTRODUCTION
Self Healing refers to the detection and correction of a software fault or failure after the problem has occurred. Although it is sometimes easy to detect a program failure it is very difficult to correct the specific problem affecting the program. The complexity of the failure may cause more time to be spent in problem determination than to actually restart the application. However, restarting the application causes disruptions to other system activities and ultimately to the end user.

Preventive measures reduce the need for more complex detection and correction methods. By preventing a large number of failures, the actual detection and correction can be reduced to a rollback and restart level rather than a more complex examination and correction approach. However, most COTS applications do not benefit from advances in the area of rollback and restart due in part to the complexity and propriety nature of the applications. Traditionally, is has not been considered necessary to endow applications such as word processors, spreadsheets, browsers and such with self healing capabilities. However, having the ability to self heal provides immense advantages for all applications, especially in mobile and distributed environments.

In our research into virtualizing operating systems, we have found the need to be able to migrate any running application, without access to its source code. This paper describes how we have been able to build migration facilities, by decoupling the application from its environment (virtualization). We then propose to extend these existing mechanisms to use in module level rejuvenation. Our approach extends a new service to legacy applications allowing them to participate in newer "seamlessly distributed" computing environments without modification to their code.

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

This research discusses both an implementation and the core work that enables existing applications to assume new and novel characteristics and behaviors. Specifically, we use virtualization and process migration technologies of the virtualizing Operating System to provide module level rejuvenation. We also describe how the *vOS* system can be used to monitor and refresh itself, thus reducing aging failure within the vOS.

[1] This research is partially supported by grants from NSF, DARPA/Rome Labs and AFOSR.

The structure of the remainder of this paper is as follows: Section 2 provides the motivation for the creation of the virtualizing Operating System. Section 3 describes architecture, components and operation of the vOS. A discussion of software aging and rejuvenation is in section 4 with preemptive module refresh covered in section 5. Our module level rejuvenation is described in section 6 with Section 7 describing the *vOS* implementation of this process. Section 8 discusses related work and section 9 provides the paper summary.

2. BACKGROUND

Our research is part of a larger project called "*Computing Communities*" (or *CC*) [1]. The goal of the *CC* project is to enable a group of computers to behave like a large community of systems. The community grows or shrinks based on dynamic resource requirements through the scheduling and moving of processes, applications and resource allocations between systems—all transparently.

The computers participating in the *CC* utilize a standard operating system and run stock applications. The key technique to achieve such a system is the creation of a *virtualizing Operating System* or *vOS* [2,3]. The main theme in the *vOS* is "virtualization", which is the decoupling of the application process from its physical environment. That is, a process runs on a virtual processor with connections to a virtual screen and virtual keyboard, using virtual files, virtual network connections, and other virtual resources. *The vOS has the ability to change the connections of the virtual resources to real resources at any point in time, without support from the application*. The *vOS* implements the functionality to virtualize the resources by controlling the mapping between the physical resources (seen by the operating system) and virtual handles (seen by the application).

Our current work is based on the Windows 2000 operating system, but is extensible to any stock operating system. Windows 2000, like other operating systems, is structured such that the applications and the operating system contain a clearly delineated point of indirection, which is easily exploitable to add or interpose a layer of middleware.

3. VIRTUALIZING OPERATING SYSTEM

The *virtualizing Operating System* (*vOS)* is the central mechanism for our research. The main *vOS* theme is "virtualization," which is the decoupling of the application from its physical environment.

3.1 Architecture

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

The *vOS* unobtrusively integrates its components into existing Windows 2000 system. It provides services that intercept and virtualize existing applications on the member machines without altering the application's coding in any way.

The bounded domain of the *vOS* system is defined through the deployment of the three main system components (figure 1): the *virtalizing System Manage (vSM)*, the *virtualizing EXecutive (vEX)* and the *virtualizing Interceptor (vIN)*. Each of these components is placed at a key control point and is responsible for contributing to the overall functionality of the *vOS*. The control

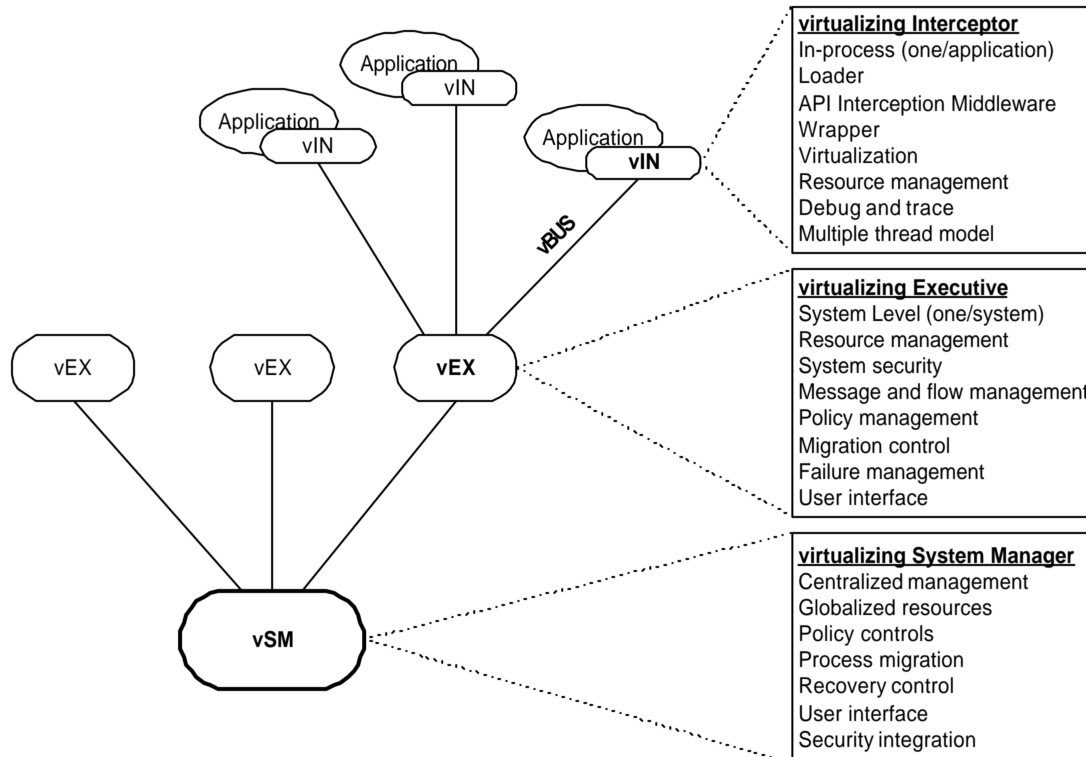

Figure 1. *vOS* Architecture

points are at either the machine or application level.

3.2 Components

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

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

The *vIN* is used by the *vEX* to initiate and capture process data from the workstation applications. The *vIN* is software injected into a running application, and it monitors the host application by intercepting its API calls to the underlying operating system. It resides in the same address space as the application under its control. It integrates a command interface into the application through the insertion of a menu item into the existing application menu, otherwise, it is completely unseen by the user of the interactive application.

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

3.3 Key Technologies

Two technologies are used to enable the operation of the vOS. The first is the API/DLL Interception, which is based on Richter's [4] description. API's are intercepted by changing the addresses contained within the Import Address Table (IAT). The IAT contains a set of pointers filled in at runtime that resolve to the API fulfillment address in the appropriate DLL. Using this approach new code is inserted between the application and the DLL. Once the API is intercepted, we capture procedure call information, such as the handle, and manipulate or change it as required.

The second technology is the creation of an abstraction layer between a running application and the underlying system through the creation and management of virtual handles. Virtual handles are values that we create to replace the original handles returned in the API by the system. The application saves and uses the replacement handles as it would the real handles. The virtual handles are connected to real handles at the time the application makes requests to the system. This approach allows an application to move between computers transparently by remapping the virtual handles to real handles when required. The application does not change nor is it aware of the underlying modifications.

These two technologies work hand in hand to collect, save and manage the API call data, reserving it in migration tables within the *vIN*. The tables become a portion of the data copied and restored during process migration or module rejuvenation.

3.4 Migration Models

In our research we determined that the task of migrating a process can be divided into three somewhat overlapping approaches based on the type of state information required. Many processes require that only a small or minimal amount of state information needs to be collected and restored; this is referred to as the Minimal State Migration Model [5]. More complex processes involving complexities such as network, file and threading require that a more complete collection and restoration of process state be undertaken. Either a Full State Migration Model is required or for more specific module level controls, an Enhanced Minimal State Migration methodology is used. The enhancements include handling of files and managing active network connections.

Nasika and Heballalu [6,7] provided the pioneering work and Zhang, Khambatti and Dasgupta [8] extended the findings for the Minimal State Model methodology. Given an application, a certain minimal set of state elements combined with a restart/suspend technique is all that is required to correctly migrate the specific set of processes. This minimal set of elements consists of the ".data" area, heap allocation and handles. Tests have shown that these three components will correctly allow the recreation of process state. If file IO is present, then methods of capturing and restoring the file state for applications are defined. If network connections are present then the middleware layer handles the connection redirection.

4. DECAY AND REJUVENATION

As software executes, it shows signs of decay or aging [9,10,11]. Decay is caused by several factors. The first is undetected design flaws that create faults and failures over time. If the program must execute for an extended period, as versus single short period task, errors begin to accumulate and failure ultimately occurs. Design flaw errors are entirely a software issue. Both poor initial designs as well as periodic code modifications are the primary culprits for causing aging to occur.

The second type of error is produced by hardware faulting. Hardware can induce the same effect as software aging. Depending on the type and degree of the fault, any prescribed software solutions may provide the same benefits. However, hardware may only be masked for a short period of time. Ultimate failures are unavoidable.

In either the software or hardware case, errors accumulate and if given enough time, will cause the program to fail. The most prescribed method to resolve software decay is software rejuvenation [11,12]. Software rejuvenation is the refreshing of program execution environment through the stopping, reloading and restarting of the program or one or more of its elements [13]. The granularity of the refreshing operation can be as fine as memory defragmentation or as large as periodically restarting the entire computer. In any case, some set of proactive actions are prescribed based on an appropriate methodology for the application.

Rejuvenation has the merit of preventing errors from occurring by reducing the amount of time that a program or one of its components actually executes. Yurcik and Doss [13] comment that rejuvenation does not, remove bugs, the approach merely dodges them very effectively. So rather than running a system for a one year and increasing the chance of a failure, why not run the system for one day, 365 times [14]. In other words, by restarting the system more frequently, fewer aging related failures will occur.

Although restarting a system periodically has a failure reduction benefit, there is a cost associated with the restart. Specifically, the cost is full system unavailability for some period of time. A more effective approach is to use some form of targeted rejuvenation. Specifically, one of the finer grain methodologies should reduce the unavailability costs.

Using the body as a model, each of the cells on the body is replaced on a periodic basis. The replacement does not cause any type of interruption or trauma. In this way, the body continually refreshes itself and reduces the possibility for failure. Thinking of the cells as programs components, this same conceptual approach can be used to target specific parts of a running system, for refreshment. The finer the targeting, the less noticeable the impact to the system caused by the refresh.

Most COTS systems are designed to be a single composite set of routines that the program thread(s) use to perform its task. The routines are not considered in isolation while they are executing even though the APIs reside in various modules. This makes refreshing at a component level a significant problem since the specific modules being used may not be known by the application. Using the *vOS*, we are able to isolate to a module level, the APIs in use at any given time. Module level is sufficient for replacement to be the least disruptive.

5. PREEMPTIVE MODULE REFRESH

The methodology, frequency and timing of software refreshment can be based on a number of approaches. In general, however, some form of preemptive replacement approach is recommended. By replacing some targeted portion of the software environment on a schedule, static frequency or algorithmically approach, the desired reduction in failures is achieved. Studies are currently working on the optimal type of approach and methodology.

One of the strategies for performing component or system refreshing is at a macro system level focusing on hardware and the operating system. The IBM approach uses clustering in conjunction with failover to perform a system level movement of an application from one cluster node to another [12]. Then the operating system from the vacated node is refreshed by rebooting. This method resolves operating system aging using the existing cluster failover algorithms. At a finer level, this approach is also able to restart an individual service. However, the restarted service must be able to handle the restart by saving and restoring the appropriate data and state information.

Gupta and Jalote [15] suggest using a reactive technique of a combination of rollback with an online change when a fault occurs. The approach requires check pointing to provide the rollback. They further suggest that a more desirable proactive approach is to correct the fault online before performing the restart. Waiting until a failure occurs is more of a backstop in fault tolerance and should be available to some degree if more active measures fail.

Garg, Huang, Kintala and Trivedi [16] recommend combining check pointing with rejuvenation, thus "reducing the amount of rollback" after a failure. The issue becomes one of when to perform the rejuvenation. We are not as concerned in this paper with when to rejuvenate, but are instead interested in the method of rejuvenation.

The approach we are recommending is finer grain than the system level, does not require hardware assistance and is not limited to specialized services or modified applications. We describe next a module level restart that avoids the expense of a clustering environment, the loss of time associated with a full system reboot and enables any COTS application to participate in the refresh process. Choosing a module level restart is more efficient and less of a performance impact. In addition, our restart approach requires no modifications to existing applications and uses the existing *vOS* capabilities.

Module level restart works by reloading a copy of the text and then reorganizing and reapplying any state information. There can be a few up to many modules associated with an active program. However, it is unlikely that the program is using every API that the module contains. In fact, although an examination of the program may indicated that it imports some number N APIs, it in fact uses some smaller number P of these APIs ($P < N$) during execution. This means that although the entire module is a target for refreshing, not every possible API carried within the module is affected. If we compare this approach to the refreshing of the entire program and all of its associated modules, the time trade off can be significant.

Our approach is to perform preemptive module refresh using the facilities of the *vOS*. The *vOS* provides an ongoing recording and storing of the state information at an application level. It saves the state information for the APIs that the application is using. This provides the best backstop for refreshing only the required APIs.

6. MODULE LEVEL REJUVENATION

The *vOS* is a middleware application that is inserted between an application and its support modules (section 3). Each of the APIs that a program uses within the support module is known by the *vOS*. The *vOS* intercepts each API call the program makes and the *vOS* code is executed before and after the API code is called.

With the API/Module information, the *vOS* performs on demand migration of active programs between systems using the *vIN*. The same data collected by the *vIN* and used for performing the migration constitutes a checkpoint and is applied to the refreshing process. One of the main distinctions between a full migration and the module level refresh is that only selected modules are refreshed. This allows for a finer refresh granularity which in turn requires less time and is less noticeable to the user.

To perform the module level rejuvenation, the *vOS* follows the following steps:

1. Suspends the program thread(s)
2. Unloads and flushes the target module
3. Reloads the target module
4. Reorganizes the module/API virtual table entries
5. Resumes the suspended thread(s)

One of the benefits of the *vOS* approach is that it does not require a specific checkpoint be performed since the data is collected as the API calls are made, thus saving time.

In the case of the application, a modified Enhanced Partial State process migration is performed (section 3.4). Instead of terminat-

ing and restoring the application and all of its module states, it only restores a selected main program component. The partial state approach allows the application's internal data to be rebuilt, essentially reorganized, before applying the final state. This allows the main program module(s) to be refreshed with minimal overhead.

7. *VOS* REFRESH

We have shown how the module level rejuvenation procedure works using the data collected and maintained by the *vIN*. The structure of the *vOS* supports further capabilities that enable additional self healing functionalities. Each of the main *vOS* components, *vSM*, *vEX* and *vIN*, are capable of differing degrees of independent operation. The *vSM*, for example, can be executed on any machine and the *vEX*s will locate and join it. The *vEX* does not require a *vSM* to operate. The *vIN* once injected by a *vEX*, operates without *vEX* involvement. Given this independence, each of the components can be extended to monitor, detect and refresh the other components.

The same aging issues apply to the *vOS* as apply to any program and it should also be required to be rejuvenated as part of the ongoing refreshing process. The individual *vOS* modules can refresh themselves as required and as a backstop, the different *vOS* components can monitor each other and resolved any issues that arise as part of the refresh process.

Since module level refresh of the *vOS* components is used, the time required to perform the refresh should be low and thus minimal impact. The *vOS* is aware of the refresh and can be readily modified to perform the refresh cooperatively. The applications are completely unaware of the refreshing process and are not affected by the activities.

Rejuvenating the operating system remains a separate issue. Regardless of the operating system in use, the primary prescribed method of refresh is through a reboot. However, even though this is required for off the shelf systems and is time consuming, it is not required as frequently as is examined in Castelli, Harper, Heidelberger, Hunter, Trivedi, Vaidyanathan, and Zeggert [12]. Since the applications are refreshed, the operating system is not required to be rebooted as frequently to resolve application aging.

8. RELATED WORK

Much of the current literature covers work related to the modeling and analysis of software rejuvenation. Our work examines the application of rejuvenation. We found work from both the industry and research sectors that apply rejuvenation technology.

BASE [17] uses an N-Version Programming replication technique combined with a layer of abstraction to hide implementation details of off-the-shelf services. It proactively uses software rejuvenation, recovering the replicas on a periodic basis and then restarting the service by "rebooting" it. Current state information is supplied by the replicas in an abstracted form and a consensus state is applied to the new copy. The thesis is that "concrete state" aging corruption is hidden through the abstracted view of the state. It uses conformance wrappers to capture state information. The service remains available to the system from one of the replicas. We have chosen to avoid the developmental overhead associated with the N-Version Programming approach. Although multiple copies of the service shows high reliability against software aging, the management and control overhead limits the applicability to COTS applications.

IBM Director [18] is a product announced by IBM (January 22, 2001) that incorporates IBM Software Rejuvenation [12]. The software proactively "identifies and predicts pending software problems" then schedules a rejuvenation for the identified software. It normally reboots the server on a planned basis or refreshes at a service level. It does not have the capability of refreshing an individual unmodified application, which is our main focus.

WinFT [19] is a set of library routines designed for Win95 and Windows NT. The routines provide support for detection and restart of a failed process, rebooting of a hung or broken operating system, checkpoint/recovery and software rejuvenation. Since the library functions must be called directly from within the application, it does not apply to existing COTS type applications.

9. SUMMARY

In this paper, we have made a proposal for performing module level replacement using the capabilities of the virtualizing Operating System. The *vOS* is the main platform implementation of the Computing Communities project and forms the basis for implementing the core technologies. It is distributed in nature and is integrated into the existing Windows 2000 environment without altering either the applications or the operating system. We believe that COTS software does not typically share the benefits of rejuvenation to reduce the effects of aging.

Software aging has been identified as a significant source of system failures. The prescribed solution to aging is software refreshing or rejuvenation. Many of the solutions available are on a system or specialized program level of granularity. We have chosen to focus on COTS applications such as spreadsheets and word processors which form the bulk of the user desktop applications. User applications stand the most to benefit from some form of aging prevention. For example, the word processor used to create this paper, crashed twice during its creation. From past experience, these crashes could have been avoided by "rebooting" the word processor on a periodic basis. Otherwise, some form of periodic module refreshing while I worked would have most likely achieved the same results.

Our focus is on preventative maintenance as a method to reduce the need for complex forms of self-healing. The module replacement approach reduces both the overhead and resultant time that a user would perceive. We have described how the capabilities of the *vOS* are tailor made for performing both module level and system level rejuvenation. No modifications are required to either the application or the operating system for our approach to work. This significantly extends the capabilities of the unchanged application and reduces the costs associated with adding the capability to existing systems.

10. REFERENCES

[1] Dasgupta, P., Karamcheti, V. and Kedem, Z. Transparent Distribution Middleware for General Purpose Computations, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), June 1999.

[2] Boyd, T. and Dasgupta, P. Virtualizing Operating Systems for Seamless Distributed Environments, in Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, vol. 2, November 2000, pp. 735-740.

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

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

[5] Boyd, T. and Dasgupta, P. Process Migration: A Generalized Approach using a Virtualizing Operating System, To be published in the 22nd International Conference on Distributed Computing Systems, Vienna, Austria, July 2002.

[6] Nasika, R. and Dasgupta, P. Transparent Migration of Distributed Communicating Processes, to appear in the 13th International Conference on Parallel and Distributed Computing Systems, August 2000.

[7] Hebbalalu, R. File Input/Output and Graphical Input/Output Handling for Nomadic Process on Windows NT, Master's Thesis, Arizona State University, August 1999.

[8] Zhang, S., Khambatti, M. and Dasgupta, P. Process Migration through Virtualization in a Computing Community, 13th IASTED International Conference on Parallel and Distributed Computing Systems, August 2001.

[9] Parnas, D.L. Software Aging, In the proceedings of the 16th International Conference on Software Engineering, May 1994.

[10] Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A. Does Software Decay? Assessing the Evidence from Change Management Data, IEEE Transactions on Software Engineering, vol. 27, no. 1, Jan. 2001.

[11] Huang, Y., Kintala, C., Kolettis, N. and Fulton, N.D. Software Rejuvenation: Analysis, Module and Applications, Proceedings of the 25th Symposium on Fault Tolerant Computer Systems, Pasadena, CA, June 1995.

[12] Castelli, V., Harper, R.E., Heidelberger, P. Hunter, S.W. K., Trivedi, S., Vaidyanathan, K. and Zeggert, W.P., Proactive Management of Software Aging, IBM Journal of Research and Development, Vol. 45 No. 2, 2001.

[13] Yurcik, W. and Doss, D. Achieving Fault-Tolerant Software with Rejuvenation and Reconfiguration, IEEE Software, July/August 2001.

[14] Berstein, L. (2002, May), A Software Engineering Course for Trustworthy Software, Available: http://www.dacs.dtic.mil/awareness/newsletters/stn3-4/trustworthy.html.

[15] Gupta, D. and Jalote, P. Increasing Software Reliability through Rollback and On-line Fault Repair, In the Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems, Taipei, Taiwan, December 1997.

[16] Garg, S. , Huang, Y., Kintala, C.M.R. and Trivedi, K.S. Minimizing Completion Time of a Program by Checkpointing and Rejuvenation, ACM SIGMETRICS 96, Philadelphia, PA, May 1996.

[17] Rodrigues, R. Castro, M. and Liskov, B. BASE: Using Abstraction to Improve Fault Tolerance, Proceedings of the 18th Symposium on Operating Systems Principles (SOSP '01), Banff, Canada, October 2001.

[18] Press Release (2002, May), IBM Delivers Next Generation of Self-Healing Management Tools, available: http://www-916.ibm.com/press/prnews.nsf/jan/B20162597ECE8738852569DC0050C6AE.

[19] Carreira, J. Costa, D. and Silva, J.G. WinFT: Using Off-the-Shelf Computers on Industrial Environments, Proceedings of the 6th International Conference on Emerging Technologies and Factory Automation, Los Angeles, September, 1997.