# Application Resilience with Process Failures

2011 International Conference on Security and Management

Special Track on Mission Assurance and Critical Infrastructure Protection

Kathleen McGill* and Stephen Taylor
Thayer School of Engineering
Dartmouth College
8000 Cummings Hall, Hanover, NH 03755
Contact Author: Kathleen.N.McGill@Dartmouth.edu
Stephen.Taylor@dartmouth.edu

tr11-006

***Abstract*—The notion of resiliency is concerned with constructing mission-critical applications that are able to operate through a wide variety of failures, errors, and malicious attacks. A number of approaches have been proposed in the literature based on fault tolerance achieved through replication of resources. In general, these approaches provide graceful degradation of performance to the point of failure but do not guarantee progress in the presence of multiple cascading and recurrent attacks. Our approach is to dynamically replicate message-passing processes, detect inconsistencies in their behavior, and restore the level of fault tolerance as a computation proceeds.**

**This paper describes a novel operating system technology for resilient message-passing applications that is automated, scalable, and transparent. The technology provides mechanisms for process replication, multicast messaging, and process failure detection. We demonstrate resilience to failures and benchmark the performance impact using a distributed exemplar representative of applications constructed using domain decomposition.**

***Keywords-resiliency; mission-assurance; distributed systems; process replication; failure detection***

## I. INTRODUCTION

Commercial-off-the-shelf (COTS) computer systems have traditionally provided several measures to protect against hardware failures, such as RAID file systems [1] and redundant power supplies [2]. Unfortunately, there has been relatively little success in providing similar levels of fault tolerance to software errors and exceptions. In recent years, computer network attacks have added a new dimension that decreases overall system reliability. A broad variety of technologies have been explored for detecting these attacks using intrusion detection systems [3]-[4], file-system integrity checkers [5]-[6], rootkit detectors [7]-[8], and a host of other technologies. Unfortunately, creative attackers and trusted insiders have continued to undermine confidence in software. These robustness issues are magnified in distributed applications, which provide multiple points of failure and attack.

Our previous attempts to implement resilience resulted in an application programming library called the Scalable Concurrent Programming Library (SCPlib) [9]-[10]. Unfortunately, the level of detail involved in programming resilience in applications compounded the already complex activity of concurrent programming. The research described here explores operating system support for resilience that is automatic, scalable, and transparent to the programmer. This approach dynamically replicates processes, detects inconsistencies in their behavior, and restores the level of fault tolerance as the computation proceeds [9]-[10]. Fig. 1 illustrates how this strategy is achieved. At the application level, three communicating processes share information using message-passing. The underlying operating system implements a resilient view that replicates each process and organizes communication between the resulting *resilient process groups*. Individual processes within each group are mapped to different computers to ensure that a single attack or failure cannot impact an entire group. The base of Fig. 1 shows how the process structure responds to attack or failure: It assumes that an attack is perpetrated against processor 3, causing processes 1 and 2 to fail or to portray communication

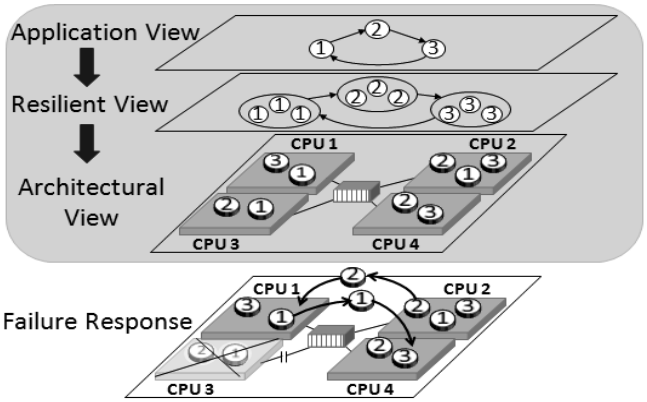


Figure 1. Dynamic process regeneration.

*This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-09-1-0213.

inconsistencies with other replicas within their group. These failures are detected by communication timeouts and/or message comparison. Detected failures trigger automatic process regeneration; the remaining consistent copies of processes 1 and 2 dynamically regenerate a new replica and migrate it to processors 4 and 1, respectively. As a result, the process structure is reconstituted, and the application continues operation with the same level of assurance.

This approach requires several mechanisms that are not directly available in modern operating systems. Process *replication* is needed to transform single processes into resilient process groups. Process *migration* is required to move a process from one processor to another. As processes move around the network, it is necessary to provide control over where processes are *mapped*. Point-to-point communication between application processes must be replaced by group communication between process groups. It is desirable to *maintain locality* within groups of replicated processes: This allows transit delays from replicated messages to be used to predict an upper bound on the delay for failure detection timeouts. Finally, mechanisms to detect process failures and inconsistencies must be available to initiate process regeneration.

This paper describes a distributed message-passing technology to support resilience. The technology has been implemented through a Linux loadable kernel module that manages resilient process groups through multicast communication and process failure detection. This module provides a minimal MPI-like message-passing API implemented through kernel TCP functions. Multicast communications provide an environment for adaptive failure detection based on process group locality. The communication module is combined with a migration module [11] to provide process mobility. Through cooperation between the communication and migration modules, the operating system can detect and dynamically regenerate failed processes of a distributed application.

To evaluate our resilient technology, we use a distributed application exemplar to represent a broader class of applications. These evaluations demonstrate application resilience to failed processes and assess the performance impact of resilience.

## II. Related Work

### A. Distributed Application Fault Tolerance

A variety of approaches have emerged to provide *fault tolerance* for distributed applications that rely on the replication of resources. These include checkpoint/restart [12]-[17] and process replication [18]-[21]. In general, these approaches provide graceful degradation of performance to the point of failure but do not *guarantee* progress in the presence of multiple cascading and recurrent attacks.

Checkpoint/restart systems save the current state of a process to stable storage for recovery. There are many distributed checkpoint/restart systems [14]-[17], [22]-[23] closely tied with the Message Passing Interface (MPI). These systems emphasize coordination of distributed process checkpoints to achieve a consistent global state of an application. In contrast, our technology does not require a global checkpoint: We use resilient process groups to provide fault tolerance, avoiding the overhead of global coordination protocols.

Process replication systems provide fault tolerance through redundant execution. P2P-MPI [19] and VolpexMPI [20] are two MPI libraries that transparently replicate MPI processes for fault tolerant execution. P2P-MPI uses a gossip-protocol to detect *node failures* so that failed nodes can be removed from the computation. VolpexMPI emphasizes performance optimization by allowing applications to progress at the speed of the fastest replica. The key difference between these approaches and ours is that these libraries use static replication. No attempt is made to dynamically recover failed processes, so multiple failures may ultimately cause application failure.

Dynamic process replication has been proposed to provide recovery of failed processes within distributed applications. MPI/FT is an MPI middleware that implements redundancy with process monitoring and dynamic process recovery [21]. Unfortunately, the approach requires a central coordinator and does not scale well. SCPLib, the predecessor to the proposed approach, provides a distributed dynamic process regeneration solution [9]-[10]. However, because SCPLib places the burden of resilience on the application programmer, it is not practical. In contrast, our approach applies these concepts in the operating system for transparent and automatic application resilience.

### B. Distributed Failure Detection

Several protocols have been proposed for distributed failure detection. Many are based on a heartbeat mechanism, in which processes periodically send heartbeat messages to indicate liveness [24]-[27]. In contrast to these approaches, we conduct failure detection using application messages. This tactic avoids the need for an additional heartbeat message. In addition, we have the capability to detect compromised processes through message inconsistencies.

Regardless of the protocol, communication timeouts are a common means for detecting failed processes. Traditionally, timeouts were determined by a fixed delay. However, Chandra and Toueg [28] demonstrated that fixed delays are unreliable because of variations in processor loads and network traffic.

More recent work in failure detection has moved to adaptive time delays. Fetzer *et al.* introduced a simple adaptive protocol in which the timeout is adjusted to the maximal delay time of prior heartbeat messages [29]. Chen *et al.* [30] and later Bertier *et al*. [31] improved on this concept by including historical message delays and network analysis to set timeouts. Unfortunately, these works revealed that the algorithms converge too slowly for reliable failure detection. Ding *et al.* proposed an alternative approach in which timeouts are determined from historical message delays alone with efficiency comparable to quality of service requirements [32]. All of these approaches rely on historical message delays to detect failures in point-to-point communications. In contrast, our approach uses the synchronized multicast messaging within process groups to provide adaptive failure detection.

Failure detection schemes within process groups have been proposed for symmetric [33] and asymmetric [34] communication configurations. Asymmetric applications are those in which a leader process is designated as the monitored process. Both approaches use *spatial multiple timeouts* in which multiple processes cooperate to monitor another process at a given time. Reliable failure detection is achieved through consensus. Our approach differs in that we use the process group multicast messaging primitives to detect failures directly. The multicast messages from process replicas serve as an adaptive baseline to detect real-time anomalies in message latency. Process group locality provides the basis for this tactic.

## III. Design and Implementation

Fig. 2 shows the software architecture of the technology. It consists of two Linux loadable kernel modules: A communication module and a migration module. These modules are implemented as character devices that provide services to user-level processes through system calls. The technology also utilizes user-level daemon processes and Linux kernel threads. The daemon processes are necessary for tasks that require a user-level address space, such as forking new processes. The message daemon forks processes to comprise distributed applications, and the migration daemon forks processes in which to regenerate failed processes. A Linux kernel thread is a process that is spawned by the kernel for a specific function. The TCP servers are kernel threads that receive incoming messages for the communication module. These servers require kernel privileges to access to the module functions and memory efficiently.

### A. Message-passing API

The communication module provides an MPI-like message-passing interface through kernel TCP functions, where a computation is a set of processes numbered from *0* to *n-1*. This minimalist API reduces the complexity of the communication state for process migration and has only three basic functions based on blocking communication:

- *msgid(&id)* --- sets *id* to be the current process identifier within the application.
- *msgsend(dest,buf,size)* – sends a message to *dest* containing the *size* bytes located at *buf*.
- *msgrecv(src,buf,size,&status)* – receives a message from *src* (or ANY) into *buf* of length *size*; *status* is a structure designating the *source* of the message and its *length*.

All functions return TRUE if successful and FALSE otherwise. In addition, as an artifact of using a character device to implement the module, two calls, *msgInitialize()* and *msgFinalize()*, are used to open and close the device and to provide a file handler for the device throughout the computation. This API can be implemented directly on top of MPI through the appropriate macros. However, the minimalist API is sufficient to support a variety of applications in the scientific community and explicitly disallows polling for messages, a major source of wasted resources. The underlying

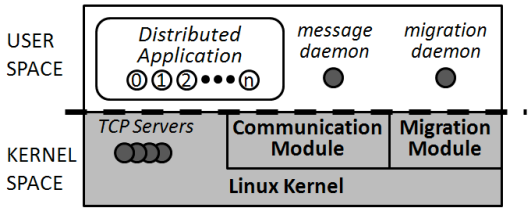


Figure 2. Software architecture of the technology.

mechanisms of the API support our resiliency model through the management of process groups, multicast messaging, and failure detection. For example, the blocking *msgrecv()* function is used to provide an environment to detect process failures and trigger regeneration.

The communication module provides four primary functions for resilient applications: application initiation, message transport, failure detection, and regeneration support.

### B. Application Initiation

A distributed message-passing program is initiated through the *msgrun* program (similar to *mpirun* used in MPI) of the form:

*msgrun* –n <n> -r <r> <program> <args>

The program takes as arguments the number of processes to spawn for the computation (*n*), the level of resiliency (*r*), and the executable program name with arguments. The *msgrun* program loads this information into the communication module, and the module sends execution information to remote modules of the cluster. At each host, the communication module signals the message daemon to fork processes for the distributed application. Recall that the applications initiated through the *msgrun* program view *n* processes total. However, in the resilient implementation, a total of *n*r* processes are forked to establish *n* process groups with resiliency *r*.

A key component of application initiation is mapping processes to hosts. To demonstrate the concepts, we use a deterministic mapping to allow each host to build a consistent process map at startup. Fig. 3 shows an example mapping of *n* process groups with resiliency *r* to *m* hosts. Processes are distributed as evenly as possible across the cluster. In order to accommodate resilient process groups, replicas are mapped to maintain locality within process groups without multiple replicas on the same host. The map is constructed by assigning replicas to hosts in ascending order. For those interested in more sophisticated mapping strategies see [35]-[39].

On initiation, an array is allocated at each communication module to store necessary information on application

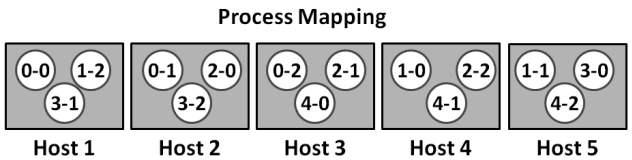


Figure 3. Mapping *n*=5 process groups with resiliency *r*=3 to *m*=5 hosts.

processes. This *process array* stores, for each process, the current mapping, the process id, the process replica number, and a *communication array* to track which hosts have sent or received messages from the process. The *msgid(&id)* call performs an ioctl() system call on the module to copy the process id and the total number of processes into process memory. Consistent with the application view of Fig. 1, application processes are unaware of replication. The total number of processes reported is actually the number of *process groups*.

### C. Message Transport

The communication module provides message transport for local and remote application messages. In the resilient model, point-to-point communication between application processes becomes multicast communication between process groups, as shown in Fig. 4. The multicast messaging protocols are transparent to the application. The *msgsend()* call performs an ioctl() on the module. For each call, the communication module iterates through a loop to send one application message to each replica of the destination process group. For each replica, the module determines whether the destination process is local or remote by referencing the *process array*. Local messages are placed on a message queue in kernel memory. Remote messages are sent via kernel-level TCP sockets to the appropriate host.

The *msgrecv()* call performs an ioctl() on the module to search the kernel message queues for the specified message. For each call, the communication module iterates through a loop to locate *r* application messages from the source process group. The first message located in a queue is copied to the designated user-space buffer. However, the *msgrecv()* call blocks until *r* messages are received. If a message is missing from the kernel queue, the module places the process on a wait queue until all messages are received or until a maximum wait time has elapsed. In this way, the *msgrecv()* function serves as an implicit synchronization point for process groups and provides a setting for failure detection.

### D. Failure Detection

The failure detection protocol is currently designed to detect process failures through communication timeouts. Within a *msgrecv()* call, a local timestamp is stored to indicate when each message is retrieved from the kernel queue. Processes that have waited the maximum time to receive messages initiate the failure detection protocol. In this protocol, two conditions indicate a process failure. First, the destination process must have already received *r-1* messages. In other words, only one message is missing. Second, the average time elapsed since the other messages timestamps must be greater than the prescribed timeout. If these two conditions are met, the destination process triggers a process regeneration for the latent message's source. In the current implementation, the timeout is set nominally at one second.

To trigger process regeneration, the destination process performs two actions. A process failure message is sent to the host of the failed process. A process regeneration message is sent to the host of the lowest live replica number of the failed process. If either the failed or replicating process is local, a signal is used instead. After triggering regeneration, the destination ignores the missing message and proceeds with the computation.


Figure 4. Point-to-point communication between application processes becomes multicast communication between process groups.

### E. Process Regeneration

The communication module provides support for process regeneration, including cleaning up after failed processes, sending the packaged process image for regeneration, and resolving communication for the regenerated process at its new location.

If a module receives a message indicating a process failure, it cleans up the process structures. If the process exists in any state, it is killed, and the process descriptor is deleted. All outgoing communication sockets for the process are closed, and any messages in the kernel queues are deleted. This approach encompasses multiple causes of failure, such as CPU and socket failures, without detecting the failure itself.

The communication module that hosts a process replica receives a message to regenerate the process. The next time the replicating process makes a *msgrecv()* call, it is interrupted for replication. The actual process replication and regeneration is accomplished by the migration module. On the host of the live replica, the migration module packages a copy of the replica process image into a buffer. The communication module selects the destination host for the regenerated process. A simple selection policy is used to prevent mapping members of the same process group to the same machine while maintaining locality. As shown in Fig. 5, the destination host selected is the nearest machine, in ascending order, unoccupied by a member of the regenerated process's group.

After the migration module has packaged the process image, the communication module executes a regeneration protocol. First, the local process map is updated to reflect the pending regeneration of the failed process at its new location. Second, update messages are sent to remote modules stored in the *communication array*. These update messages contain the process id, the process replica number, and the new host to update remote process maps. In addition, the update messages

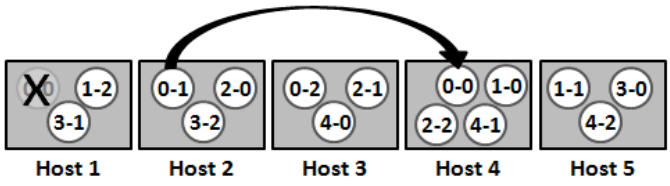


Figure 5. Dynamic process regeneration: Process 0-0 is migrated to host 4.

allow those processes waiting for messages from the failed process to progress until regeneration is complete. Third, all messages for the replicating process in the kernel message queue, are duplicated and forwarded to the destination host for the regenerated process. Finally, the process image is sent to the destination host.

When the packaged process image is received by the destination host, the migration module restores the process image. The regenerated process resumes execution by repeating the original *msgrecv()* call that was interrupted for regeneration. For more detail on the process replication and migration mechanisms, see [11].

Application messaging for failed processes may occur simultaneously within the regeneration protocol. Messages for the regenerated process may arrive at its original host after it has failed or at the destination host before the process update messages are received. Two features of the technology guarantee that these messages eventually reach the destination process: automatic message forwarding and reactive process update messages. The communication module forwards messages for failed processes after regeneration and sends reactive updates to ensure the remote process map is revised. These features enable guaranteed message delivery for failed or migrated processes without global coordination protocols. For more detail on the communication module support for messages-in-transit during process migration, see [11].

## IV. Experimental Evaluation

To evaluate the technology, preliminary benchmarks were conducted using a domain decomposition exemplar. These benchmarks serve two purposes. The first is to evaluate performance impact of the resilient message-passing technology. The second is to demonstrate application resilience in the presence of process failures.

The Dirichlet exemplar is a simple problem taken from fluid dynamics that is representative of a wide class of applications solved with domain decomposition [40]. It involves a large, static data structure and uses an iterative numerical technique over a cellular grid that converges to the solution of Laplace's Equation. Our parameterization solves the Dirichlet problem using a two-dimensional decomposition in which nearest-neighbor dependencies are resolved through communication with neighboring partitions.

The primary metrics used to assess performance are the average execution time of the application and the overhead of resilience. The overhead of resilience is the percentage increase in average execution time of applications with replication. The metrics used to demonstrate application resilience are associated with the completeness and accuracy of failure detection. Completeness refers to the percentage of failures that are detected, and accuracy refers percentage of false positive detections. In addition, we measure the overhead of process failures as the difference in average execution time of resilience applications with and without failures.

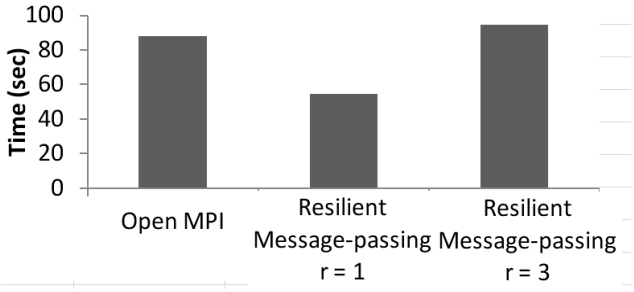


Figure 6. Average execution times using Open MPI, using resilient message-passing without replication (r = 1), and using resilient -assing with triple resiliency (r = 3).

These benchmarks were executed on a dedicated Dell PowerEdge M600 Blade Server with 8 hosts. Each host has dual Intel Xeon E5440 2.83GHz processors and 16 GB of memory. The hosts are connected by a 1 Gbps Ethernet network. The operating system is Ubuntu 10.04.01 LTS Linux 2.6.32-26 x86_64. The cluster also has Open MPI v. 1.4.1 for comparison with a standard message-passing system. For each test, 32 processes, or process groups, were spawned.

### A. Experiment 1: Resilient Message-passingPerformance

The performance of the resilient message-passing technology was evaluated to assess the impact of process replication. In addition, the performance was compared to Open MPI to ensure that the technology does not incur prohibitive overhead. Fig. 6 shows the average execution times of the exemplar using Open MPI, using resilient message-passing without replication ($r$ = 1), and using resilient message passing with triple resiliency ($r$ = 3). Without replication, the resilient MP outperforms Open MPI. The Open MPI execution time is 61% longer than resilient message-passing.

The overhead of triple resilience is approximately 73%. This overhead is a result of both multicast communications and redundant computations. In multicast messaging with triple resiliency, a single application message corresponds to 9 resilient messages. The resilient process groups in these benchmarks result in a total of 96 processes running on 64 cores, overloading the system resources. These results are consistent with the priority placed on resilience in the technology development. No optimizations have been made which could compromise resilience.

### B. Experiment 2: Application Resilience with Failures

To demonstrate application resilience, the exemplar was executed with process failures. In each execution of the application, one process was manually killed from the shell console. Fig. 7 shows the average execution times of the

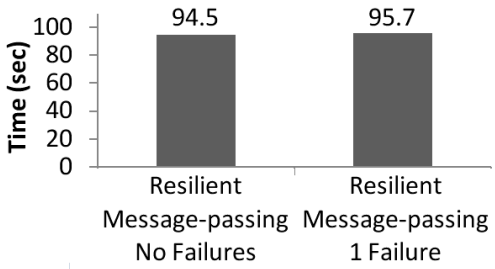


Figure 7. Average execution times using triple resiliency with and without a single failure.

exemplar using triple resiliency with and without a single failure. In each test, the application operates through the process failure in order to complete the computation. The overhead of a process failure is 1.2 seconds, which is less than 2% of the execution time. This overhead is attributed to the one second timeout to detect the process failure and the duration of the process regeneration protocol.

The failure detection protocol detects 100% of process failures in these tests. As a result, each failed process is dynamically regenerated, and the application continues with reconstituted resilience. In addition, there are no false positives detected in the failure tests or during the benchmarks of the technology without failures.

## V. Conclusions and Future Work

This paper describes a resilient message-passing technology for mission-critical distributed applications. The technology is comprised of operating system support for resilience. The communication module is a minimal MPI alternative that provides a resilient message-passing API, detects process failures, and resolves message transport for regenerated processes. The migration module packages a process image into a kernel buffer, sends the buffer to a remote machine, and restores the process execution at the new location. This technology achieves application resilience to failures automatically and transparently without global communication.

This technology is designed for applications that prioritize resilience over performance. The performance of resilient message-passing *without failures* is quantified using a distributed exemplar that represents mission-critical applications. The technology incurs 73% overhead in execution time with triple resiliency. For those mission-critical applications which cannot tolerate the performance impact, the target system must be supplemented with additional compute resources to reduce the overhead due to computational redundancy.

Application resilience is also demonstrated in the presence of failures. The failure detection algorithm detects all process failures with no false positives. This result enables dynamic process regeneration, so the application operates through failures to restore triple resiliency. The approximate overhead of process failure is 1.2 seconds for a single failure of the exemplar application.

This technology serves as a demonstration of our approach to application resilience. In the future, we plan to explore alternative failure detection algorithms and add capability to detect compromised processes through message comparison. In addition, while we have seen evidence of resilience to simultaneous failures, we have not benchmarked these events in a controlled test. We seek to perform extensive benchmarking using multiple application exemplars, larger scales, and multiple cascading and recurring failures.

## Notice

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

## References

[1] D.A. Patterson, G. Gibson, and R.H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management Data*, 1988, pp. 109-116.

[2] Lefurgy, X. Wang, and M. Ware, "Server-level power control," in *Proceedings of the International Conference on Autonomic Computing*, 2007, pp. 4-13.

[3] R. Di Pietro and L.V. Mancini, *Intrusion Detection Systems*, New York, Springer-Verlag, 2008.

[4] Singhal, "Intrusion Detection Systems," in *Data Warehousing and Data Mining for Cyber Security*, vol. 31, pp. 43-47, 2007.

[5] G.H. Kim and E.H. Spafford, "The design and implementation of tripwire: A file system integrity checker," in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994, pp. 18-29.

[6] J. Kaczmarek and M. Wrobel, "Modern approaches to file system integrity checking," in *Proceedings of the 1st International Conference on Information Technology*, 2008.

[7] N.L. Petroni, T. Fraser, J. Molina, and W.A. Arbaugh, "Copilot- a Coprocessor-based Kernel Runtime Integrity Monitor," in *Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 179-194.

[8] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008, pp. 1-20.

[9] J. Lee., S.J. Chapin, and S. Taylor. "Computational Resiliency", *Journal of Quality and Reliability Engineering International,* vol. 18, no. 3, pp 185-199, 2002.

[10] J. Lee, S. J. Chapin, and S. Taylor, "Reliable Heterogeneous Applications," IEEE Transactions on Reliability, special issue on Quality/Reliability Engineering of Information Systems, Vol. 52, No 3, pp. 330-339, 2003.

[11] K. McGill and S. Taylor, "Process Migration for Resilient Applications," *ACM Symposium on Parallelism in Algorithms and Architectures*, unpublished.

[12] Hua Zhong and Jason Nieh, "CRAK: Linux Checkpoint / Restart As a Kernel Module", Technical Report CUCS-014-01, Department of Computer Science, Columbia University, November 2001.

[13] P. Hargrove and J. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters" In *Proceedings of SciDAC 2006*: June 2006.

[14] V. Chaudhary and H. Jiang, "Techniques for migrating computations on the grid," In *Engineering the Grid: Status and Perspective*, B. Di Martino et al., Eds. American Scientific Publishers, 2006, pp. 399 – 415.

[15] Gengbin Zheng, Chao Huang, Laxmikant V. Kale, Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++, *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems, 2006*.

[16] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," in LACSI, Oct. 2003.

[17] J. Hursey, J.M. Squyres, T.I. Mattox, and A. Lumsdain, "The design and implementation of checkpoint/restart process fault tolerance for OpenMPI," In Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium, March 2007.

[18] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt and D.A. Connors, "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance," *In Proceedings of the 37th IEEE/IFIP*

*International Conference on Dependable Systems and Networks*, Edinburgh, UK. June 25-28, 2007.

[19] S. Genaud and C. Rattanapoka, "P2P-MPI: A peer-to-peer framework for robust execution of message passing parallel programs on grids," *Journal of Grid Computing,* Vol 5, pp 27-42, 2007.

[20] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok. "VolpexMPI: An MPI Library for Execution of Parallel Applications on Volatile Nodes." In Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Helsinki, Finland, September, 2009.

[21] Batchu, R., Neelamegam, J., Cui, Z., Beddhua, M., Skjellum, A., Dandass, Y., Apte, M.: MPI/FTTM: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel. In: Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid. Melbourne, Australia (2001).

[22] G. Bosilca, A. Boutellier, and F. Cappello, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in Supercomputing, Nov. 2002.

[23] Agbaria, A., Friedman, R.: Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing. Los Alamitos, California, pp. 167–176 (1999).

[24] M. Pasin, S. Fontaine, and S. Bouchenak, "Failure detection n large scale systems: a survey," *In Proceedings of the IEEE Network Operations and Management Symposium Workshops*, July 2008.

[25] C. Dobre, F.. Pop, A. Costan, M Andreica, and V. Cristea, "Robust failure detection architecture for large scale distributed systems," *Proc. of the 17th Intl. Conf. on Control Systems and Computer Science*, pp. 433-440, May 2009.

[26] M. K. Aguilera, W. Chen, and S. Toueg, "Heartbeat: a timeout-free failure detector for quiescent reliable communication," *In Proceedings of 11th International Workshop on Distributed Algorithms*, pp. 126-140, September 1997.

[27] R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," *In Proceedings of International Conference on Distributed Systems Platforms and Open Distributed Processing,* 1998.

[28] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM, 43*(2), pp. 225-267, March 1996.

[29] C. Fetzer, M. Raynal, and F. Tronel, "*An adaptive failure detection protocol*," *In Proceedings of the 8th IEEE Pacific Rim Symp. on Dependable Computing*, pp. 146--153, 2001.

[30] W. Chen, S. Toueg, and M. Aguilera, "On the quality of service of failure detectors," *IEEE Trans. on Computers, 51*(5), pp. 561- 580, 2002.

[31] M. Bertier, O. Marin,and P. Sens, "Implementation and Performance Evaluation of an Adaptable Failure Detector", *in Proc.of the 15th Int'l Conf. on Dependable Systems and Networks*, 2002, pp. 354-363.

[32] X. Ding, Z. Gu, L. Shi, Y. Hou, and L. Shi, "A failure detection model based on message delay prediction," *In Proceedings of the IEEE International Conference on Grid and Cooperative Computing*, Lanzou, China, 2009.

[33] I. Gupta, T. Chandra, and G. Goldszmidt, "On scalable and efficient distributed failure detectors," *In Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing,* pp. 170, 2001.

[34] X. Li and M. Brockmeyer, "Fast Failure Detection in a Process Group," In Proceedings of the Parallel and Distributed Computing Symposium, 2007.

[35] A. Heirich and S. Taylor "Load Balancing by Diffusion", Proceedings of 24th International Conference on Parallel Programming, vol 3 CRC Press pp 192-202, 1995.

[36] J. Watts, and S. Taylor, "A Vector-based Strategy for Dynamic Resource Allocation", Journal of Concurrency: Practice and Experiences, 1998.

[37] K. McGill and S. Taylor, "Robot algorithms for localization of multiple emission sources," *ACM Computing Surveys (CSUR)*, in press.

[38] K. McGill and S. Taylor. "DIFFUSE algorithm for robotic multi-source localization", *2011 IEEE International Conference on Technologies for Practical Robot Applications*, Woburn, Massachusetts, April 2011, in press.

[39] K. McGill and S. Taylor. "Operating System Support for Resilience", *IEEE Transactions on Reliability*, unpublished.

[40] Taylor and Wang, "Launch Vehicle Simulations using a Concurrent, Implicit Navier-Stokes Solver", AIAA Journal of Spacecraft and Rockets, Vol 33, No. 5, pp 601-606, Oct 1996.