Operating System Support for Resilience

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

by

Kathleen McGill

Thayer School of Engineering

Dartmouth College

Hanover, New Hampshire

August 2011

Examining Committee:

Chairman_____
Stephen Taylor, Ph.D.

Member_____
Peter Haaland, Ph.D.

Member_____
George Cybenko, Ph.D.

Member_____
Eugene Santos, Ph.D.

_____
Brian W. Pogue
Dean of Graduate Studies

Abstract

The notion of resiliency is concerned with constructing applications that are able to operate through a wide variety of computer failures and attacks. Several approaches have been proposed to provide fault tolerance through the 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 failures. The proposed approach dynamically replicates processes, detects inconsistencies in their behavior, and restores the level of resiliency as the computation proceeds, so that failures have no long term effect on applications. This thesis introduces a collection of novel operating system technologies that provide applications with automated, transparent, and scalable resilience. Resiliency *mechanisms* and *policies* are explored in a resilient message-passing technology, *rMP*.

A Linux rMP prototype implements a message-passing API through kernel-level communication that provides the underlying resiliency mechanisms: process replication, adaptive failure detection, and dynamic process regeneration. An innovative approach to adaptive failure detection uses locality within replicated processes as a basis to detect anomalies in message delay during group communication. Replication and migration mechanisms provide transparent regeneration of message-passing processes without halting the application or executing global coordination protocols.

Resiliency policies are explored through evaluation of alternative algorithms for distributed process management. A new algorithm, DIFFUSE, is introduced, inspired by the notions of *heat diffusion* and *robotic swarming*. Heat diffusion is emulated to

disseminate processes across a scalable multicomputer architecture. Robotic swarming techniques are used to *maintain locality* between resilient processes while balancing load. The algorithm's performance is compared to competing algorithms using a set of benchmarks that capture the primary attributes of the process management problem. DIFFUSE outperforms competing algorithms and integrates the goals of load-balancing and resilience within a single strategy.

Acknowledgements

I would like to acknowledge the people who have supported me in my studies at Dartmouth. Special thanks to my advisor Dr. Stephen Taylor for his guidance in this research and in my career transition. I am grateful to have a mentor with such a remarkable combination of educational expertise, career insight, and well-balanced life view to encourage me. I would also like to thank Dr. Peter Haaland for his contributions to my dissertation research and my career development. Peter's counsel was invaluable to my exploration and pursuit of career opportunities after graduation. Thanks also to the remaining members of my dissertation committee, Dr. George Cybenko and Dr. Eugene Santos. Their contributions to this project have enhanced the quality and significance of my research in the field.

I would like to thank my research group at Dartmouth: Steve Kuhn, Colin Nichols, Morgon Kanter, and Mike Henson. Through numerous discussions, presentations, and casual insights, these colleagues have increased my knowledge and understanding of a wide variety of topics in the fields of computer science and engineering. In addition, our group camaraderie provides support throughout the achievements and set backs of graduate education. Additional thanks to the staff at the Thayer School of Engineering, especially Karen Thurston, for her continuous help in navigating the administrative and logistic details of pursuing a Ph.D.

I would like to acknowledge the Defense Advanced Research Projects Agency (DARPA) for their sponsorship of this project. This research is sponsored under agreement number FA8750-09-1-0213.

Most significantly, I would like to thank my husband, John. His support, encouragement, and tremendous patience have been invaluable these last few years. I appreciate the positive outlook that he enforces during the excitement and occasional frustration of our post-graduate pursuits. I would also like to thank my parents, James and Ann McGill, as well as John's parents, John and Kate Cavanaugh. They have motivated John and me to pursue higher education and to apply our skills for the greater good. Finally, I would like to thank our many brothers, sisters, nephews, and nieces, who have provided outstanding motivation and entertainment over the years.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

This thesis is concerned with distributed applications that execute on multiple processors and share information through message-passing. Today, distributed applications are unreliable because they cannot operate through computer failures and attacks. Unfortunately, it is unlikely that we will ever be able to prevent, or even detect, all of these events. Thus, the primary goal of this thesis is to allow distributed applications to be *resilient* to failures and attacks.

At the same time, computer systems are increasing in scale across the board—the number of processors per computer, the amount of memory, the speed and capacity of networks, etc. Knowing this, *scalable* resiliency solutions are the objective so that distributed applications will scale with available technology.

These goals pose two key questions: (1) How do we design a computer system that will operate through failures and attacks even if they are never detected? (2) How do we automate process management of distributed applications in a scalable manner?

1.1 Motivation

Commercial-off-the-shelf (COTS) computer systems have traditionally provided several measures to protect organizations from hardware failures, such as RAID file systems [1] and redundant power supplies [2]. Unfortunately, there has been relatively little effort to provide 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]-[5], file-system integrity

checkers [6]-[7], rootkit detectors [8]-[11], 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.

A variety of approaches have emerged to provide reliability for distributed applications that rely on the static replication of resources. These include checkpoint/restart [12]-[16], process migration [17]-[23], and process replication [24]-[28]. In contrast, computational resiliency provides reliability through *dynamic* replication of resources [28]-[30]. The Scalable Concurrent Programming Library (SCPlib) [30], developed in the late 1990's, was an early proof-of-concept for distributed and dynamic process replication. SCPlib was a library that allowed application programmers to build resilient process groups and enable process failure detection with dynamic process regeneration. However, the main conclusion from this research was that resiliency is too complicated for application programmers. To be practical, the concepts needed to be provided to distributed applications *automatically* and *transparently*.

Figure 1 displays the relative impact of resiliency and static replication on the reliability of applications under attack [30]. Static replication alone provides graceful performance degradation to the point of failure because each attack reduces the number of replicas permanently. In contrast, resiliency dynamically reconstitutes the desired number of process replicas after each attack, allowing the application to continue operation with the same level of assurance. The malicious actions have no long term effect.

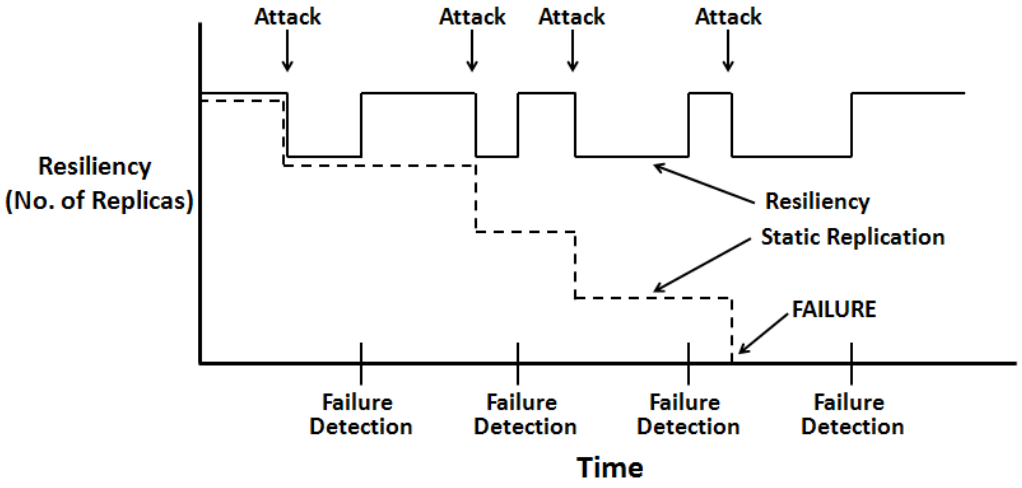


Figure 1. Reliability of applications using computational resiliency and static replication [30].

1.2 Approach

The approach of this thesis builds on the concept of resilience introduced in the SCPlib research [30] but provides *automatic* and *transparent* resilience for applications. This approach is achieved by implementing resiliency mechanisms within the *operating system*. A collection of novel operating system technologies have been designed to dynamically replicate processes, automatically detect inconsistencies in their behavior, and transparently restore the level of resiliency as the computation proceeds. Figure 2 illustrates how this strategy is achieved [30]. At the application level, three processes share information using message-passing. The underlying operating system directly 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 failure or attack cannot impact an entire group.

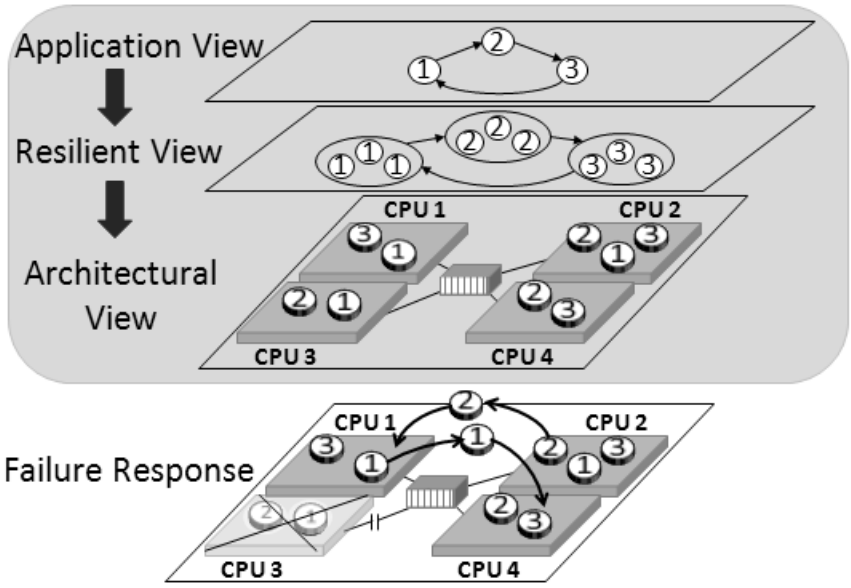


Figure 2. Dynamic process regeneration

The base of the figure shows how the process structure responds to failure or attack [30]. The figure assumes that an attack is perpetrated against processor 3, causing processes 1 and 2 to fail or to portray communication inconsistencies with other replicas within their group. Failures are detected by communication timeouts and message comparison. These 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, process resiliency is reconstituted, and the application continues operation with the same level of assurance.

In order to achieve this approach, several features are required that are not directly available in modern operating systems. Process *replication* is needed to transform single processes into process groups. Point-to-point communication between application processes must be replaced by group communication between process groups. Mechanisms to detect process failures and inconsistencies must be available to initiate

process *regeneration*, and process *migration* is required to move a process from one processor to another. Finally, as processes move around the architecture, it is necessary to provide control over where processes are *mapped*. In order to prevent prohibitive communication costs, process management policies are desired to *maintain locality* within process groups. This tactic enables *locality-based failure detection*, in which transit delays from replicated messages are used to predict an upper bound on the delay for communication timeouts.

These basic resiliency capabilities add complexity to process management. Scalable solutions require distributed and automated implementation. Process scheduling algorithms must meet both resiliency and performance requirements: (1) Maintain locality between processes in the same process group. (2) Map process replicas to different processors. (3) Distribute the load across the system.

1.3 Contributions

This thesis describes novel operating system technologies that provide resilience for distributed applications. The significant contributions concern resiliency *mechanisms* and *policies* associated with a resilient message-passing technology, *rMP*. The resiliency mechanisms achieve process replication, adaptive failure detection, and dynamic process regeneration automatically and transparently within an operating system. Resiliency policies are explored through design and evaluation of alternative algorithms for distributed process scheduling. Finally, an analytical framework in introduced to enable concrete reliability analysis of the proposed approach to resilience. The following are the unique contributions of this thesis:

- A Linux prototype of the rMP technology provides a resilient application programming interface (API) as a minimalist alternative to the Message Passing Interface (MPI). The API is implemented through a kernel-level communication module that provides applications with automated resiliency mechanisms [31].
- A collection of failure detection algorithms are based on adaptive communication timeouts and message comparison [31], [32], [33]. Adaptive failure detection uses process group locality as a basis to detect anomalies in message delay during group communication. Comparison of replicated messages allows detection of process inconsistencies through majority voting.
- Replication and migration mechanisms enable transparent regeneration of message-passing processes. Regeneration is accomplished by blending prior work in Linux-based migration with the distributed communication support of the rMP technology [32].
- The problem of non-determinism in resilient applications is introduced and explored. Non-deterministic processes pose a significant challenge for replication technologies. A preliminary solution is proposed that preserves resiliency for rMP applications.
- An analytical framework is presented that includes a collection of performance benchmarks and analyses that allow the overhead of resilience to be evaluated [33].
- Resilient process management policies are explored through a comparative analysis of robotic swarming algorithms for distributed process scheduling. Swarming algorithms enable distributed processes to achieve a common goal

while imposing *swarm cohesion* (i.e. locality between members of a swarm). The basis for the comparison is a novel set of benchmarks with an associated sensitivity analysis that capture the primary attributes of the process management problem [34], [35].

- A novel DIFFUSE algorithm is presented, inspired by the notions of *heat diffusion* and *robotic swarming* [33], [36]. Heat diffusion is emulated to distribute processes across scalable computer architecture [37], [38]. Robotic swarming techniques are used to maintain locality between replicated processes. This work combines concepts from both perspectives to integrate the goals of resilience and performance in a single strategy.
- A second analytical framework is developed to perform reliability analysis of the rMP design with respect to common fault and threat models.

1.4 Scope

In this thesis, resilience is defined as the ability to provide and maintain an acceptable level of service in spite of a range of faults and attacks [39]. This definition is a question of ongoing debate. Often, the term is considered a synonym of fault tolerance [40]. However, resiliency researchers argue that fault tolerance allows systems to operate in the presence of *isolated* failures. Resiliency goes further to provide system *recovery* and operation through *multiple* failures to *maintain* the original level of service [39]. To describe the application level of service, this thesis also defines the *level of resiliency* as the number of replicas in the resilient implementation.

In order to provide application resilience, this research is concerned with *process failures*. Two modes of failures are addressed: complete failures and Byzantine failures.

A complete failure is when a process halts, and the external state is constant [40]. The process ceases communication with other processes of the application. A Byzantine failure, on the other hand, is a failure in which the process behaves in an unpredictable manner [41]. Byzantine failures encompass a range of failures, such as the omission of some action, corruption of local state, or inconsistent communication [40], [41].

There is no attempt to detect or diagnose the root cause of process failures in this work. Processor faults, network failures, software exceptions, and malicious events are all treated the same. As a result, the mechanisms used in the approach are not tailored to a particular fault or threat model. These models are only considered in the context of reliability analysis in order to quantify the reliability of the approach.

1.5 Thesis Organization

The remainder of this thesis is organized as follows.

- Chapter 2 presents related research in resilience and fault tolerance technologies.
- Chapter 3 describes the rMP technology, including the failure detection and process regeneration mechanisms.
- Chapter 4 explores the problem of non-determinism in resilient applications.
- Chapter 5 presents performance benchmarks and analysis of the rMP technology.
- Chapter 6 presents a survey of strategies for resilient process management.
- Chapter 7 provides ground-truth comparative analysis of candidate algorithms for the process management problem.
- Chapter 8 presents a novel DIFFUSE algorithm and a quantitative analysis of its process scheduling performance.

- Chapter 9 explores reliability of the approach in a number of fault and threat models.
- Chapter 10 summarizes the limitations of the current research, areas for future work, and the main conclusions of this research.

Chapter 2 Related Research in Distributed Application Resilience

A variety of approaches have emerged in the literature to provide fault tolerance for distributed applications that rely on the replication of resources. These include checkpoint/restart [12]-[16], process migration [17]-[23], and process replication [24]-[28]. A survey of the research in these fields is provided, with an emphasis on their relationship to the rMP technology.

2.1 Checkpoint/restart

Checkpoint/restart (C/R) systems save the current state of a process to stable storage for recovery and provide the underlying mechanisms for process replication and migration. Several solutions have been posed at the user-level [14] and kernel-level [12], [13], [15], [16]. Kernel-level solutions have two key advantages. They have kernel privileges to capture all features of the process state, allowing the checkpoint of fully featured processes. They also operate transparently to the application, enabling use for unmodified applications.

There are many distributed checkpoint/restart systems closely tied with the message-passing libraries of applications [14]-[16]. These systems emphasize the coordination of distributed process checkpoints to achieve a consistent *global checkpoint* of an application [42]. In contrast, the rMP technology does not require a global checkpoint: Resilient process groups are used to ensure fault tolerance, avoiding global coordination protocols altogether.

2.2 Process Migration

Process migration is the movement of a process from one host to another, and has been a well-researched topic for over 20 years [17]. In the context of resiliency, process regeneration is a two step procedure of process replication and migration; a process is replicated, and the *replica* is migrated. The fundamental mechanisms for regeneration and migration are the same, allowing migration research to be leveraged in this work.

Migration systems must incorporate C/R mechanisms in order to save the state of a process. The challenge in migrating message-passing processes is to guarantee message transport during and after migration. To achieve this, the communication state of a process must be integrated into the C/R mechanisms. This research focuses on kernel-level solutions and builds on prior work in Linux-based migration [20]-[23].

Kerrighed is a Linux-based distributed operating system that provides a single system image through a distributed shared memory (DSM) model and dynamic communication streams [20], [43]. The DSM simplifies process migration because a process can continue to access memory on its original host after migration. However, this capability creates *residual dependencies*. A residual dependency occurs when a migrated process relies on a resource of the original host after migration. For example, portions of the process address space remain on the original host *after migration* until they are accessed by the process. These dependencies are not resilient because a failure of the original host can cause the migrated process to fail if it attempts to retrieve this memory. The proposed approach in rMP differs from Kerrighed in that all dependencies are removed after migration.

Cruz [22]-[23] is a Linux-based system that migrates entire message-passing applications. It uses a thin virtualization layer to abstract system resources. Two features of Cruz are particularly appropriate for this work because they preserve transparency: (1) Process checkpoints are performed from outside the process context, and (2) checkpoints target the lowest level process features, such as TCP sockets rather than the message passing interface (MPI). The rMP technology utilizes these two concepts to preserve transparent migration but provides fine grain migration of individual processes rather than entire applications.

LAM-MPI [15] is an MPI library that integrates the Berkeley Lab Checkpoint Restart (BLCR) system [13] to enable migration for MPI processes. The migration of individual MPI processes faces the challenge of resolving transport for messages that are in-transit during migration. This challenge is complicated by the fact that BLCR checkpoints are conducted at the kernel level, but the communication state is maintained at the user level in the MPI interface. This problem requires a method for the kernel-level system to access user-level state without compromising transparency to the application. The LAM-MPI solution employs coordinated checkpoints for migration. User-level hooks in the BLCR system notify the MPI library of a pending checkpoint. This notification triggers a distinct sequence of actions: computation is suspended, message channels are drained, the process migrates, the message channels are re-established, and the computation resumes. Essentially, this approach disallows in-transit messages by *halting the entire application for a single process migration*. This global constraint incurs unnecessary overhead in order to retrofit the MPI library for fault tolerance. In rMP, this constraint is removed by implementing a kernel-level

communication module as an alternative to MPI. Message transport is resolved at the kernel level concurrently with process replication and migration. This fine-grained strategy allows applications to progress in the presence of failures and malicious actions without global coordination.

2.3 Process Replication

Process replication is an alternative strategy that enables fault tolerance through redundant execution. Several MPI libraries have emerged that transparently replicate MPI processes for fault tolerance [25]-[27]. P2P-MPI [25] targets grid computing with fluctuating configurations, using replication to provide robustness to machine failures. Similarly, rMPI [26] executes replicated MPI processes on redundant hosts to facilitate fault tolerance at extreme large scale. VolpexMPI [27] emphasizes fault tolerance with minimal performance impact by forcing applications to progress at the speed of the fastest replica. The key difference between these approaches and rMP is that these libraries use static replication. No attempt is made to recover failed processes, so multiple failures may ultimately cause application failure.

Dynamic process replication has been proposed to enable recovery of failed processes within applications. MPI/FT is an MPI middleware that implements redundancy with process monitoring and dynamic process recovery [28]. Unfortunately, the approach requires a central coordinator and does not scale. SCPlib, the predecessor to rMP, is a distributed and dynamic process regeneration solution which provides application programmers with library calls to construct resilience [30]. However, because SCPlib places the burden of resilience on the application programmer, it is not

practical. In contrast, the rMP technology applies these concepts in the operating system for transparent and automatic resilience.

2.4 Distributed Failure Detection

A crucial component of dynamic process replication is the detection of failed processes. Two failure detection methods are leveraged in rMP: adaptive communication timeouts for complete process failures and message comparison for Byzantine process failures.

2.4.1 Communication Timeouts

Several protocols have been proposed for distributed failure detection through communication timeouts. Many are based on a heartbeat mechanism, in which processes periodically send heartbeat messages to indicate liveness [44]-[47]. In contrast to these approaches, this work conducts failure detection using application messages. This tactic avoids the need for an additional heartbeat message.

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 [48] demonstrated that fixed delays are unreliable because of variations in processor loads and network traffic.

More recent work in failure detection has produced a progression of adaptive time delay algorithms. Fetzer *et al.* introduced a simple adaptive protocol in which the timeout is adjusted to the maximum delay time of prior heartbeat messages [49]. Chen *et al.* [50] and later Bertier *et al*. [51] 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 [52]. All of these approaches rely on historical message delays to detect failures in point-to-point communications. In contrast, the rMP approach uses synchronized multicast messaging within process groups to provide adaptive failure detection.

Failure detection schemes within process groups have been proposed for symmetric [53] and asymmetric [54] 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. The rMP approach differs in that it uses the process group multicast messaging primitives to detect failures directly. The replicated messages serve as an adaptive baseline to detect real-time anomalies in message latency. Preserving process group locality provides the basis for this tactic.

2.4.2 Message Comparison

Byzantine failures are often thwarted through replication techniques. Computations are replicated; if there is disagreement in the results, majority voting is used to determine a consensus on which result is correct [55], [56]. Traditional attempts at Byzantine fault tolerance assume identical and synchronized replicas, in which all replicas are voting on the *exact* same results in the same order. Under these conditions, majority voting provides Byzantine fault tolerance to f faulty processes with $2f+1$ replicas [57].

However, realistic applications often involve asynchronous communications, in which message transmission contributes to uncertainty in Byzantine agreement. In particular, the transmission can cause unpredictable message delays or fail in the middle of an execution. Fischer *et al.* have shown that in an asynchronous model, Byzantine agreement becomes impossible to achieve in bounded time with even one faulty process [58]. This demonstration hinges on message delays that compromise the order in which messages are received. Several early solutions to Byzantine failures are variations on majority voting to accommodate asynchrony [55], [58], [59], [60], [61]. For example, the Rampart toolkit uses tightly controlled messaging protocols to provide reliable transmission and to convert asynchronous communication into "virtually synchronous" communication for Byzantine agreement [55]. Similarly, in rMP, messaging primitives are designed to be independent of the application and provide reliable message transport. In addition, the rMP technology tracks the order in which messages are *sent* to ensure that voting is conducted on the *same message*. These measures preserve Byzantine failure detection with $2f+1$ replicas.

A recent resurgence in this area aims to improve the efficiency of Byzantine fault-tolerant algorithms [62]-[66]. The argument is that early systems are too slow to be practical, due to redundant computations and communications. In contrast to these systems, these redundancies are already built into the rMP technology. As a result, the simplicity of majority voting in group communication outweighs the benefit of optimization [57].

2.5 Summary and Conclusions

The rMP technology leverages progress in the fields of checkpoint/restart, process migration, process replication, and distributed failure detection. However, rMP is unique in that it includes all of these mechanisms in a comprehensive solution; none of the existing systems meet all the criteria of resilience. The best practices of each field are integrated with the rMP technology to provide a single strategy, tailored to resilience.

Chapter 3 The rMP Technology[1]

The rMP technology is composed of two Linux loadable kernel modules. Figure 3 shows the software architecture of the technology. The *communication module* is a MPI-alternative that provides the rMP concurrent message-passing application programming interface (API) and resolves communication for regenerated processes [31]. The *migration module* performs process regeneration by copying a process image into a kernel buffer, sending the buffer to a remote machine, and restoring the process execution at the new location [32].

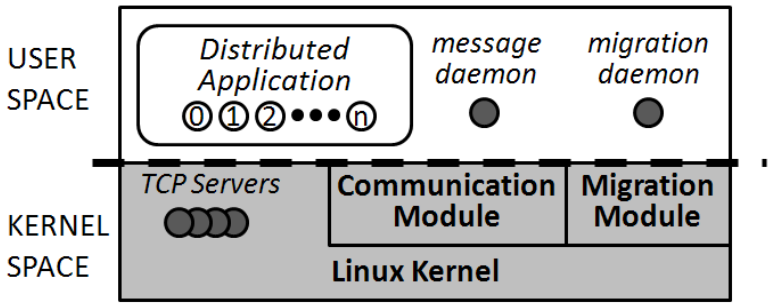


Figure 3. Software architecture of the technology.

The Linux kernel modules are implemented as 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

[1] Significant portions of this chapter are published in the following:
- K. McGill and S. Taylor, "Application Resilience with Process Failures," *Proc. of 2011 Int. Conf. on Security and Management*, Las Vegas, NV, July 2011.
- K. McGill and S. Taylor, "Computational Resiliency for Distributed Applications," *MILCOM 2011*, Baltimore, MD, Nov 2011, accepted for publication.

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 the functions and memory of both modules efficiently.

3.1 Technology Base

The communication module serves as the local manager of resilient applications. It provides the core functions for the message-passing API, application initiation, and message transport of resilient applications.

3.1.1 Message-passing API

The rMP API reduces the complexity of the communication state for process migration and regeneration 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.

In addition, as an artifact of using a device to implement the module, *msgInitialize()* and *msgFinalize()* calls 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 a simple set of macros. However, the minimalist API is sufficient to support a variety of applications in the scientific and military communities. The API explicitly disallows polling for messages, a major source of wasted resources in message-passing systems such as MPI. This functionality can instead be achieved by multi-threading. The underlying mechanisms of the API enable the resiliency model through the management of resilient process groups, multicast messaging, and failure detection.

The decision to use the rMP API in place of MPI was carefully considered. MPI has become the standard for message-passing applications. As a result, the research community has made significant effort to retrofit MPI for fault tolerance through distributed checkpoints, process migration, or replication. In contrast to these systems, rMP includes both replication and migration as core properties. This dual capability is unique in the field and would require a major re-engineering of existing MPI libraries and systems to enforce policies that are not supported by the MPI standard [67]. Thus, the design decision was to implement low-level support for the preferred API, rather than transform an existing MPI library.

To illustrate usage of the rMP API, two distributed application algorithms are presented. Figure 4 presents a concurrent algorithm for a Numerical Integration application. The application exhibits the primary characteristics of a problem solved with functional decomposition: an algorithm is decomposed into components that are computed independently without reference to a particular data structure (e.g. Climate Modeling, Floor-plan Optimization) [68]. The parameterization of Numerical Integration

decomposes the integral over an interval and sums the integration of each component using a simple ring messaging pattern. In each iteration, all processes compute an interval of the integration (12). The process with id = 0 initiates communication by sending its result to the next process in the ring (14). The processes with id > 0 wait for a message from the previous process in the ring (23), add their local result (24), and pass the sum to the next process (25). Process 0 receives the final result (15), computes the global error (16), and sends the error to each process of the application (19). This iteration continues until the global error is less than a threshold value.

```
int main(int argc, char **argv) {                                                    1
    int i,id,n,next,prev,strips, t, done, src, size;                                 2
    double a,b, mypart, result, lastresult, error;                                   3
                                                                                     4
    msgInitialize(&argc,&argv,n);              /*Initialization for rMP application*/  5
    msgid(&id);                                                                      6
    next=(id+1)%n;                                                                   7
    prev=(id+n-1)%n;                                                                 8
    get_args(argc,argv,&a,&b,&strips);                                               9
                                                                                     10
    for(error=HUGE_VAL,done=FALSE, t=1; !done; t++) {                                11
      mypart=integrate(id,n,a,b,strips);                  /*Everyone compute */    12
      if(id==0) {                                         /*Process 0*/            13
          msgsend(next,&mypart,sizeof(mypart));           /*Send result to ring*/  14
          msgrecv(prev,&result,sizeof(result), &src, &size);  /*Recv result from ring*/  15
          error = fabs(lastresult - result);              /*Compute global error*/ 16
          lastresult = result;                                                       17
                                                                                     18
          for(i=1;i<n;i++)                                /*Send global error*/    19
              msgsend(i,&error,sizeof(error));                                       20
      }                                                                              21
      else {                                              /* Process id > 0 */      22
          msgrecv(prev, &result, sizeof(result), &src, &size);  /*Recv result from ring */  23
          result += mypart;                               /*Add local sum */       24
          msgsend(next, &result, sizeof(result));         /*Pass it on */          25
          msgrecv(0, &error, sizeof(error), &src, &size);  /*Receive error */      26
      }                                                                              27
      if(fabs(error)<THRESHOLD || t==MAX)                 /*Termination Check*/    28
          done=TRUE;                                                                 29
      else                                                                           30
          strips = strips*2;                                                         31
    }                                                                                32
    msgfinalize();                                /*Finalization for rMP application*/  33
}                                                                                    34
```

Figure 4. Numerical Integration concurrent algorithm pseudo-code.

Figure 5 presents a concurrent algorithm for an application solving the Dirichlet problem. The Dirichlet problem represents problems solved with domain decomposition applications in which the problem is decomposed over a large, static data structure (e.g. Fluid Dynamics, Computational Chemistry) [69]. The application uses an iterative numerical technique over a cellular grid that converges to the solution of Laplace's

Equation. This parameterization solves the Dirichlet problem using a two-dimensional decomposition in which dependencies are resolved through nearest-neighbor communication. Each process begins by initialing its block of the domain (8). In each iteration, a process sends its block edges to its neighbors (13-16) and receives adjacent block edges from its neighbors (19-22). Each process performs the numerical iteration over its block (24). Processes with id>0 send the local norm of their iteration to the process with id = 0 and receive the global norm in return. This iteration continues until the global norm is less than a threshold value.

```
int main(int argc, char **argv) {                                                    1
    int n,id,t,done, i, j, src, nsize, ssize, esize, wsize;                          2
    double localnorm,lastnorm, gnorm,recvnorm, normsize;                             3
    Blockp bp;                                                                       4
                                                                                     5
    msgInitialize(&argc,&argv,n);          /*Initialization for rMP application*/  6
    msgid(&id);                                                                      7
    bp=initializeBlk(argc,argv,n,id);      /*Initialize process block data structure*/ 8
                                                                                     9
    for(lastnorm=HUGE_VAL, done=FALSE, t=1; !done;  t++) {                           10
                                                                                     11
      /* Send block edges to neighbors*/                                             12
      msgsend(north(bp),northbuf(bp), isize(bp));          /*to north*/             13
      msgsend(south(bp),southbuf(bp), isize(bp));          /*to south*/             14
      msgsend(east(bp),eastbuf(bp), jsize(bp));            /*to east*/              15
      msgsend(west(bp),westbuf(bp), jsize(bp));            /*to west*/              16
                                                                                     17
      /* Receive block edges from neighbors*/                                        18
      msgrecv(north(bp),nrecvbuf(bp),commsize(bp),&src,&nsize);    /*from north*/    19
      msgrecv(south(bp),srecvbuf(bp),commsize(bp),&src,&ssize);    /*from south*/    20
      msgrecv(east(bp),erecvbuf(bp),commsize(bp),&src,&esize);     /*from east*/     21
      msgrecv(west(bp),wrecvbuf(bp),commsize(bp),&src,&wsize);     /*from west*/     22
                                                                                     23
      localnorm = block_iteration(bp);    /* Iterate over block and compute a local norm*/ 24
                                                                                     25
      /*Compute global norm of iteration*/                                         26
      if(id(bp)==0) {                                  /*Process 0 */              27
         gnorm=localnorm;                              /*Computes global norm*/    28
         for(i=1; i<n; i++) {                                                        29
            msgrecv(i,&recvnorm,sizeof(recvnorm),&src,&normsize);                    30
            gnorm+=recvnorm;                                                         31
         }                                                                           32
         gnorm=gnorm/n;                                                              33
         for(i=1; i<n; i++)                            /*Sends globalnorm to id > 0*/ 34
            msgsend(i,&gnorm,sizeof(gnorm));                                         35
      }                                                                              36
      else {                                                 /*Process id > 0*/      37
         msgsend(0,&localnorm,sizeof(localnorm));            /*Send localnorm to 0*/ 38
         msgrecv(0,&gnorm,sizeof(gnorm),&src,&normsize);     /*Recv globalnorm*/     39
      }                                                                              40
                                                                                     41
      if(fabs(gnorm-lastnorm)<THRESHOLD || t==MAX)         /*Termination Check*/     42
         done=TRUE;                                                                  43
      lastnorm=gnorm;                                                                44
    }                                                                                45
    msgfinalize();                                 /*Finalization for rMP application*/ 46
}                                                                                    47
```

Figure 5. The Dirichlet Problem concurrent algorithm pseudo-code.

The Numerical Integration and Dirichlet Problem algorithms demonstrate parallel programming techniques used to construct deterministic behavior in distributed applications. There are several sources of non-determinism in computers systems, such as variations in processor speeds and contention in the communication network. These variables impact the speed at which processes of an application progress as well as the order in which messages are delivered in the network. The communication patterns in the Numerical Integration and Dirichlet Problem are constructed to enforce the order in which messages are received by each process, regardless of the order in which messages are delivered in the network. For example, the ring messaging pattern of the Numerical Integration requires that processes with id > 0 receive a message from the previous neighbor, send a message to the next neighbor, and receive the global error from process id = 0. This pattern creates a synchronization barrier for processes at each iteration. Similarly, in the Dirichlet problem, processes send messages to nearest neighbors, receive messages from nearest neighbors, send a message to process id = 0, and receive a message in return. The ordering of rMP API calls enforces the order in which messages are received and provides synchronization of processes in spite of underlying non-determinism in the system.

3.1.2 Application Initiation

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

msgrun –n <n> -r <r> <program> <args>

The program takes as arguments the number of processes in the computation (*n*), the level of resiliency (*r*), and the executable program name with arguments. For

example, a Dirichlet Problem application is initiated with 16 processes and triple resiliency using: *msgrun* –n 16 -r 3 dirichlet 4 4 1000 1000. The program arguments stipulate a decomposition of a 4000 x 4000 unit domain into 4 x 4 blocks of 1000 x 1000 units. The *msgrun* program loads this information into the communication module, and the module sends execution information to remote modules of the distributed architecture. At each host, the communication module signals the message daemon to fork processes for the distributed application. 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 prototype the technology, a deterministic mapping is used to allow each host to build a consistent process map at startup. Figure 6 shows an example mapping of *n* process groups with resiliency *r* to *m* hosts. Processes are distributed as evenly as possible across the computer architecture. 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 successive hosts in ascending order. This simple mapping strategy is achieved through modular arithmetic on the process id and replica number, defined by (1).

$$H = (r * process_{id} + process_r) \,\%\, m + 1 \qquad (1)$$

In (1), H is the host number, and $\%$ is the modulus operator. The resilient process management policies explored in this thesis offer more sophisticated mapping strategies for the future.

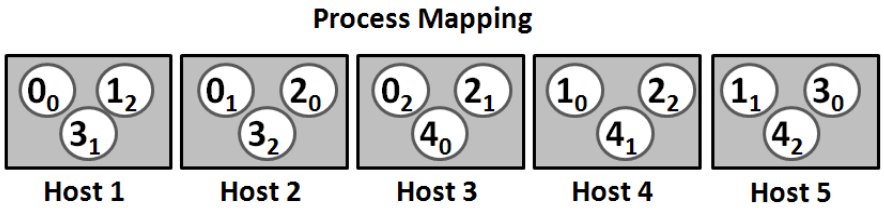


Figure 6. Mapping $n = 5$ process groups with resiliency $r = 3$ to $m = 5$ hosts.

On initiation, an array is allocated at each communication module to store necessary information on application 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. When processes call the *msgid(&id)* function, an ioctl() system call is performed on the module to copy the process id and the number of process *groups* into process memory.

3.1.3 Message Transport

The communication module provides message transport for local and remote application messages. Recall that in the resilient model, point-to-point communication between application processes becomes multicast communication between process groups, as shown in Figure 7. The multicast messaging protocols are implemented in the module, transparently to the application.

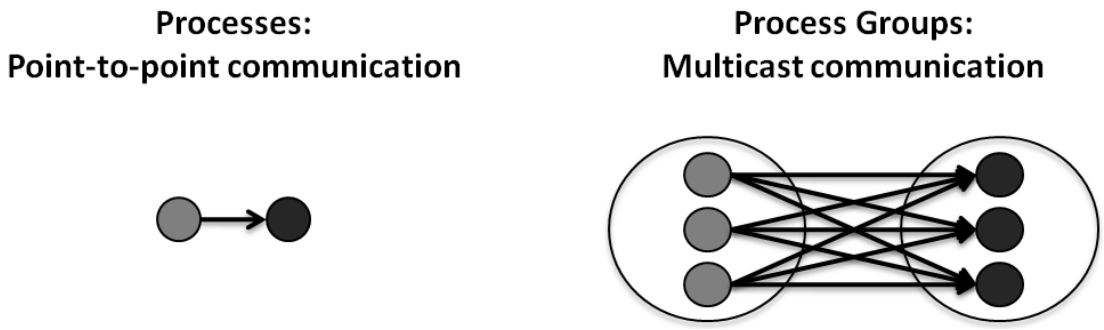


Figure 7. Point-to-point communication between application processes becomes multicast communication between process groups.

The *msgsend()* call performs a system call on the module. A simplified pseudo-code of the *msgsend()* algorithm is presented in Figure 8. For each call, the module iterates through a loop to send one message to each replica of the destination process group (1). For each replica, the module determines the resilient process id from the specified message destination and the iteration number (2). The module determines whether the destination process is local or remote by referencing the *process array* (3). Local messages are placed on a message queue in kernel memory (4), and remote messages are sent to the appropriate host via kernel-level TCP sockets (6).

```
msgsend(dest,buf,size)

for(i= 0; i<r; i++)  {                                  /*1*/
     res_id = resilient_id(dest, i);                    /*2*/
     if(host(res_id) == local)                          /*3*/
          put_msg_in_queue(res_id, buf, size);          /*4*/
     else                                               /*5*/
          remote_send(res_id, buf, size);               /*6*/
}
```

Figure 8. Pseudo-code of *msgsend()* multicast algorithm.

The communication module stores messages for local processes in multiple kernel message queues implemented using the standard Linux structures [70]. A hash table is used to organize the queues. Figure 9 displays the layout of the hash table and the message structures stored in each queue. Each hash table entry points to the head of a message queue with single a spinlock for that queue. The lock is only required for queue modifications—adding or removing an element. To locate the desired message queue, the table index can be computed with the message destination process id using modular arithmetic: table_index = process_id % TABLESIZE. This hash table reduces contention for exclusive access to the queues during message transport.

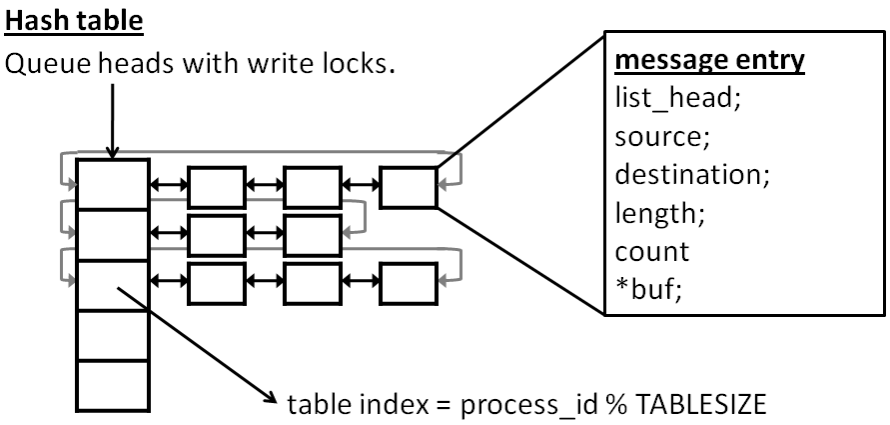


Figure 9. Hash table layout to store multiple message queues for an application.

The message structure includes the identifying parameters for the application message. The *list_head* is a requirement of the Linux linked list implementation. The *source* and *destination* parameters indicate the resilient id of the communicating processes. The *length* and *buf* parameters indicate the size and temporary location of the message contents in kernel memory. The *count* parameter is used to order messages from a process group. The same *count* is applied to each message of a replicated *msgsend()* call to ensure that the order of message delivery is preserved and that failure detection algorithms are applied to the appropriate messages.

Remote application messaging is implemented with persistent sockets created on-demand by the communication module on the message source host. The module manages a *socket array* that stores dedicated sockets for each pair of communicating processes. To send an outgoing message, the module first refers to the socket array. If a connected socket exists, it is used. Otherwise, a new socket is created and connected for the message. The new socket address is stored in the array for the remainder of the computation.

On the destination host, concurrent TCP servers manage incoming messages. When the communication module is loaded, it spawns the main TCP server to listen for new connections. For each connection request, the TCP server creates a new socket, accepts the connection, and spawns a new TCP server kernel thread to receive messages over the socket for the duration of the computation. When application messages are received, the TCP server places them on the kernel message queues. However, if the message is received with a destination process that is not local, the message is automatically forwarded to the appropriate host.

The *msgrecv()* call performs a system call on the module to search the kernel message queues for the specified message. A simplified pseudo-code of the *msgrecv()* call is presented in Figure 10. For each call, the module iterates through a loop to locate *r* messages from the source process group (2). Similar to the *msgsend()* call, the resilient process id is computed at each iteration (3). For each id, the kernel queue is traversed to locate the appropriate message (4). The module copies the contents of the first message that is located to the process's user-space buffer (6). The *msgrecv()* call blocks until all *r* messages are received. If a message is missing from the kernel queue, the module places the process on a wait queue until it is received or until a maximum wait time has elapsed (8). In this way, the *msgrecv()* function serves as an implicit synchronization point for process groups and provides a setting for failure detection (9).

```
msgrecv(src,buf,size,&status)

msgs_recvd = 0;
while(msgs_recvd < r)  {                                    /*1*/
    for(i= 0; i<r; i++)  {                                  /*2*/
        res_id = resilient_id(src, i);                      /*3*/
        for_each_msg_in_queue  {                            /*4*/
            if((res_id == msg->src)){                       /*5*/
                receive_and_compare_msg();                  /*6*/
                msgs_recvd++;                               /*7*/
            }
        }
    }
    wait_for_messages();                                    /*8*/
    faiure_detection();                                     /*9*/
}
```

Figure 10. Pseudo-code of *msgrecv()* multicast algorithm.

3.1.4 Summary of Technology Base

Application initiation and message transport represent the minimal requirements of the rMP technology. Alone, these functions provide the rMP API and enable failure-free execution of applications. This foundation is extended to enable failure detection and process regeneration for application resilience.

3.2 Failure Detection

Process failure detection is achieved within the *msgrecv()* call. Adaptive communication timeouts are incorporated into the blocking mechanisms of the algorithm, and message comparison is integrated as messages are retrieved from the kernel queues.

3.2.1 Adaptive Failure Detection

To enable communication timeouts, a temporary timestamp is stored in the communication module to indicate when each message is retrieved from the process's

kernel message queue. The adaptive failure detection protocol is invoked for a process that sleeps too long while waiting for messages. In the current 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 from the group. Second, the process must have waited longer than the adaptive timeout.

The general definition of the adaptive communication timeout is the following:

$$T = C_{AVG} t_{AVG} + C_{COMP} t_{COMP} + C_{DELAY} \quad (2)$$

In (2), T is the adaptive timeout, C_{AVG} is a communication constant, t_{AVG} is the average latency of the first *r-1* messages, C_{COMP} is a computational constant, t_{COMP} is the computational interval, which is the interval between subsequent *msgrecv()* calls with the designated source process, and C_{DELAY} is an additional constant delay. The first term in (2) accounts for the message latency, the second term accounts for variations in compute time, and the last term inserts additional padding in the timeout.

Several parameters of (2) have been explored in prior work [31], [32], [33]. The progression of these experiments is described qualitatively to explain the current definition of the adaptive timeout. The evolution of the equation provides accommodation of different classes of distributed applications.

A constant adaptive timeout was explored in which a nominal one second delay was used ($C_{AVG} = 1$, $C_{COMP} = 0$, $C_{DELAY} = 1$) [33]. This corresponds to a simple adaptive detection policy: If the destination process waits one second longer for a message than the average wait for the initial messages, it triggers process regeneration. Note that although a constant delay is used, the approach is locality-based and adaptive because the delay is calculated relative to the initial messages received from the group. In contrast, a

fixed delay would include only the constant term ($C_{AVG} = 0$, $C_{COMP} = 0$, $C_{DELAY} = 1$). The constant adaptive timeout was sufficient for a limited set of application tests.

This strategy was optimized to provide a variable adaptive timeout [32]. For a range of conditions, the C_{AVG} and C_{DELAY} parameters were varied to tune the performance of the failure detection algorithm. The goal of this investigation was to minimize the time to detect process failures without incurring false positives. This extension emphasizes the locality-basis of the communication timeout by increasing the impact of the average message latency and minimizing the size of the constant delay. While the variable adaptive timeout outperformed the constant adaptive timeout, it was not versatile enough for a wide representation of distributed applications.

Optimization of the failure detection algorithms revealed that the performance depends on the granularity of the application. In parallel computing, granularity refers to the ratio of computation to communication in applications. Coarse-grained applications have a relatively high ratio. At the extreme, this is characterized by large amounts of computation and relatively infrequent communication. These applications portray greater variations in message latency within a single process group than fine-grained applications. The computational term of the adaptive timeout was added to be more robust to coarse-grained applications [33]. The term increases the adaptive timeout for these applications with minimal impact on the timeout of fine-grained applications. The algorithm provides reliable failure detection for all three distributed application exemplars used to benchmark the performance of the rMP technology as described in Chapter 5.

3.2.2 Message Comparison

A key advantage of the rMP technology is that message comparison can be performed on replicated messages at little additional cost. This capability enables verification of messages and detection of Byzantine process failures, or *compromised* processes through computer network attack.

Message comparison is achieved within the *msgrecv()* call by applying a simple voting policy to identify inconsistent messages. Recall that when the module iterates to receive *r* messages from the source process group, it optimistically copies the contents of the first message received into the user buffer. To facilitate voting, it also initializes a vote parameter to one. Subsequent messages that are located in the queue are compared to the first message using the Linux *memcmp()* function. If the message matches the first message, the vote is incremented, and the message is discarded. If a message differs, it is stored in a temporary buffer, and the vote is decremented.

After one inconsistent message is received, subsequent messages are compared to both the user-space buffer and the temporary buffer. Messages that match the user buffer increment the vote while messages that match the temporary buffer decrement the vote. After all *r* messages have been received, the sign of the vote indicates which message is anomalous. If the first message is identified as the culprit, the contents of the user space buffer must be replaced with the temporary buffer. The current rMP prototype assumes only two distinct messages are possible and reports an error otherwise.

3.2.3 Triggering Process Regeneration

Regardless of the failure mode, each failure detection triggers process regeneration. The communication module sends two messages: a process failure

message to the host of the failed process and a process regeneration message to the host of the lowest live replica of the failed process. After triggering regeneration, the process that detected the failure ignores the anomaly and proceeds with the computation.

3.3 Process Regeneration

The rMP technology provides process regeneration through a two-step replication and migration of a live process replica. To accomplish this, it is necessary to copy the live process state which consists of the memory, kernel, and communication states. The memory state includes the process address space and CPU register contents. The kernel state includes all kernel data structures associated with the process and any system services in use, such as open files. The communication state includes any communication links with other processes and any pending messages.

Regeneration of resilient processes is achieved through cooperation between the migration and communication modules. The migration module conducts the actual regeneration by copying and packaging a process state into a buffer and restoring it to execution at the new location. The communication module provides a number of supporting functions, such as cleaning up after failed processes, sending the packaged process image to its destination host, and resolving the communication state for the regenerated process. By leveraging the distributed mechanisms of the communication module, regeneration is achieved without halting the application or executing global coordination protocols.

3.3.1 Linux Data Structures

The rMP mechanisms are tailored to the Linux data structures. Figure 11 shows a simplified diagram of the data structures for each process in the Linux kernel. A unique

descriptor called a *task_struct* contains all the necessary process information. It includes a pointer to a memory descriptor, *mm_struct *mm* that represents the process address space. The *mm_struct* descriptor includes a pointer to a linked list of virtual memory area structures, *vm_area_struct *mmap*. This list describes all memory areas that are in use by the process. The *mm_struct* also contains a page global directory pointer, **pgd*, which is an index to the process page tables. The *task_struct* also stores information on the filesystem (*fs_struct*), open files (*files_struct*), signals (*signals_struct*), and more. The migration module saves a minimal process state for migration, which include the virtual memory, the register contents, the *task_struct*, the *mm_struct*, the *vm_area_struct* list, the *files_struct*, all open files, and some open devices.

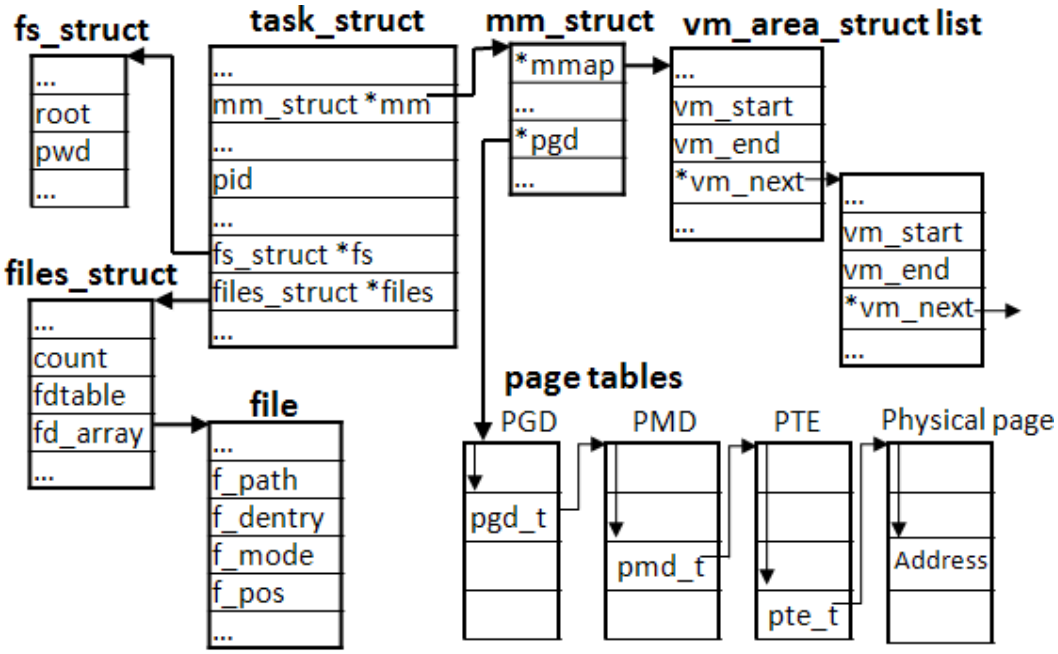


Figure 11. Linux kernel data structures for a process.

3.3.2 Process Save

When a communication module receives a message to regenerate a process, it interrupts the local process replica during its next *msgrecv()* call. The communication module selects the destination host for the regenerated process. As a demonstration of concept, 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 Figure 12, the destination host selected is the nearest machine, in ascending order, unoccupied by a member of the regenerated process's group. In the future, the resilient management policies described in this thesis will be used for host selection.

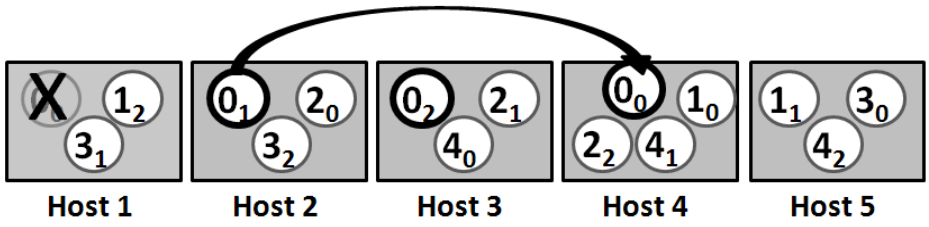


Figure 12. Process 0_0 is regenerated on host 4.

After a process is interrupted for replication, it sleeps while the migration module saves its state from outside the process context. This tactic requires some manual procedures to be conducted by the migration module that would normally be provided by the core operating system. However, it allows the module to initiate regeneration without modifications to user-level libraries or signal handlers.

The migration module performs diskless checkpointing, saving the process image to memory rather than writing it to disk. This technique is permissible because the image is only required for process regeneration; it is not intended as a stable backup. This

method avoids the overhead of writing images to disk, which is a common bottleneck in the checkpoint literature.

The process save begins with the memory state. The module walks the required memory of the process by traversing the *vm_area_struct* list shown in Figure 11. However, because the process is out of context (i.e. not running), the memory contents cannot simply be copied by the module. For each page of virtual memory, the page tables are traversed to locate the physical address of the page. If the page is not resident in memory, it is manually swapped into kernel memory before it is copied to the process buffer. To save the communication state, the migration module only saves the open file descriptor associated with the communication module. The remainder of the communication state is addressed by the communication module. Finally, the kernel state is captured in the structures described in Figure 11.

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 that have communicated with the failed process at least once. These update messages contain the process id, the process replica number, and the new host to update remote process maps. Third, all messages for the local replica in the kernel message queues are duplicated and forwarded to the destination host for the regenerated process. Finally, the process image is sent to the destination host.

3.3.3 Process Restore

On the destination host, the process is restarted using a custom exec-like function. When a process image is received, the migration daemon forks a new process which

immediately calls the migration module. The module replaces the forked process image with the buffered process image. The process restore operation uses existing Linux functions for most of its tasks. The Linux *do_unmap()* and *do_mmap()* functions are used to erase the inherited address space and rebuild the address space from the process image. The CPU context is restored by updating the register contents. Finally, the module reopens any files and devices for the process. Recall that the process image was saved during a *msgrecv()* call. The instruction pointer and system call registers are manipulated to force the regenerated process to repeat the original *msgrecv()* call on the new host when execution resumes.

3.3.4 Failed Process Clean-up

When a communication module receives a message indicating a local process failure, it cleans up the process structures to avoid conflict with the regenerated process. If the process exists in any state, it is killed. Any outgoing communication sockets for the process are closed, and any messages remaining in the kernel queues are deleted. This approach intentionally removes the failed process from the application, regardless of the process's state when the failure message is received. It is designed to encompass multiple causes of failure, such as CPU or socket failures, without detecting the failure itself.

3.3.5 Messages In-Transit

Application messaging for failed processes may occur simultaneously with 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. The following scenarios illustrate how the delivery of messages simultaneous with migration is resolved.

Figure 13 illustrates a scenario where a message is already in-transit at the beginning of process regeneration. Host A is the original hosts of the failed process. Host B is the host of the process replica used for regeneration. Host C is the destination host for the regenerated process. The scenario begins when the application message is sent from host B (1). At approximately the same time, the process regeneration protocol begins on host B. The host B process map is updated, and remote hosts are messaged with the process update (2). Hosts A and C receive the update messages and update their local maps (3) and (4). Host A receives the message for the failed process from host B, but the process is no longer local (5). Host B forwards the application message to host C (5) and sends a reactive process update message to host B (6). Subsequent application messages from host B are sent directly to host C (7). This scenario involves two extra messages: the forwarded application message in (5) and the reactive process update message in (6).

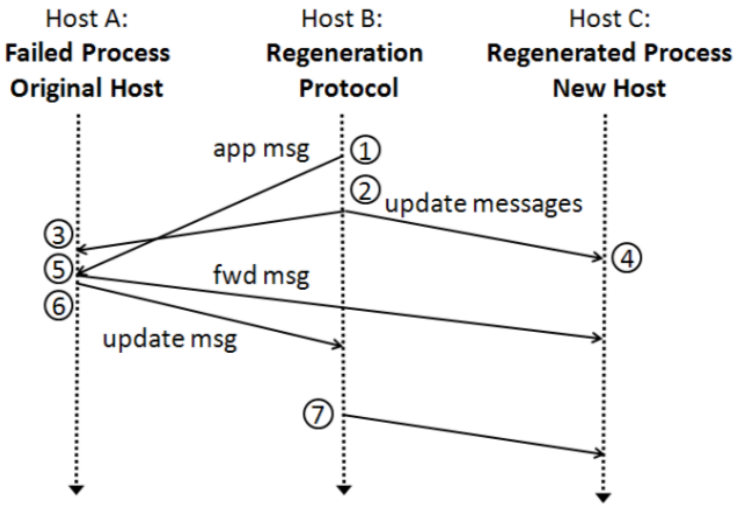


Figure 13. Regeneration scenarios: a message in-transit when process regeneration occurs.

Figure 14 illustrates a second scenario where the destination host receives application messages for the regenerated process before its process map is updated. This scenario begins with the regeneration protocol. Host B updates its own process map and sends process update messages to hosts A and C (1). Host B then sends an application message for the regenerating process to host C (2). Host C receives the application message from host B (3) before it receives the process update from host B (5). Host C automatically forwards the application message to host A (4). Host A receives the forwarded message from host C (7) and responds by forwarding the message *back* to host C and sending a reactive process update message (8). This scenario involves three extra messages: the forwarded application messages in (4) and (7) and the reactive process update message in (8).

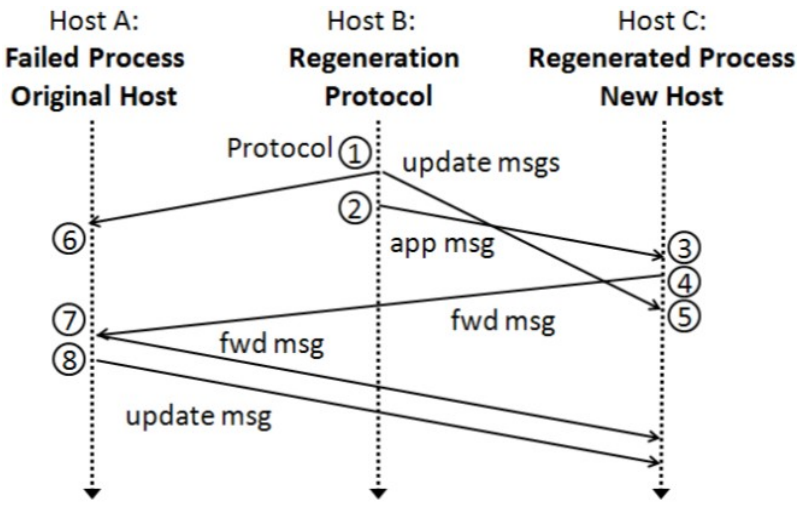


Figure 14. Regeneration scenarios: a message simultaneous with migration reaches the destination host before the process update message is received.

The extra messages in both scenarios represent the costs of guaranteed message delivery without global coordination protocols. Note that in both scenarios, the reactive process update messages are sent to hosts that have already received updates through the regeneration protocol. For simplicity, the actions of forwarding messages and sending reactive updates are automatically conducted by the TCP servers. The servers do not perform logic about the destination of reactive updates.

3.4 Summary of Contributions

The rMP technology provides the core mechanisms of resilience. The main contributions include the rMP API, kernel-level messaging protocols, locality-based failure detection, and process regeneration.

The implementations of failure detection and process regeneration represent a relatively minor extension of the base rMP technology. This fact reinforces the value in re-thinking message-passing systems to enable simpler support for resiliency. By

deviating from the MPI standard, development efforts are focused on exploring resilient mechanisms within the rMP base rather than revising MPI libraries to accommodate new features. This emphasis reduces the obstacles to realizing novel strategies for resilience.

Chapter 4 Resilience and Non-determinism[2]

There are several sources of non-determinism in computers and networks, such as variations in processor speeds, contention in the communication network, etc. Recall the distributed application examples presented in Chapter 3, in which communication is organized to provide barriers and process synchronization. These techniques construct deterministic outcomes on top of non-deterministic variables. However, some features of message-passing systems allow inherently non-deterministic applications. Many implementations include wildcard receive operations that allow a process to receive a message from any process in a given group. This is achieved through the use of a wildcard parameter, e.g. the ANY parameter of rMP or MPI_ANY_SOURCE of MPI. Messages are often received in the order that they arrive. As a result, non-determinism in the network can impact a process state. This capability is included in rMP because it is essential in dealing with irregular applications.

4.1 Example of a Non-deterministic Distributed Application

There are many cases in which non-determinism is useful in distributed applications. Consider parallel programs in which *irregular* decomposition techniques are used. In these problems, the data structures and algorithms evolve at run-time and the problem is dynamically partitioned (e.g. Adversarial Modeling, Game playing, Partial Dynamics, Cloud Computing) [71]-[74]. A common approach is to use a manager-worker communication scheme to delegate partitions of work to processes of the

[2] Portions of this chapter have been submitted for publication in the following:
- K. McGill and S. Taylor, "Operating System Support for Resilience," *IEEE Transactions on Reliability*, submitted for publication.

application. Wildcard receive parameters are used by the manager to send a partition to any worker that requests it.

Figure 15 shows an example of a basic manager-worker communication scheme. The manager begins by initializing a work list (1) and the number of active workers (2). The manager conducts the following sequence. It receives work requests from the workers using a wildcard operation (4). As long as there is work available (6), the manager responds by selecting a partition from the work list (7) and sending it to the worker (8). When there is no more work on the list, the manager answers work requests with a 'done' message (9) and decrements the number of active workers (10). Often, the work requests include data or results from previous partitions, and the manager performs some processing of results in the interim (5) or at the end (11).

Manager Code		**Worker Code**	
create_work_list(work_list);	/*1*/	work_done = FALSE;	/*1*/
active = number_of_workers;	/*2*/		
		while (!work_done) {	/*2*/
while(active) {	/*3*/	msgsend(manager, request, size);	/*3*/
msgrecv(ANY,…);	/*4*/	msgrecv(manager,response,…);	/*4*/
process_work();	/*5*/		
if(work_available) {	/*6*/	if(response == partition)	/*5*/
partition=select(work_list);	/*7*/	do_work(partition);	/*6*/
msgsend(src, partition, size);	/*8*/	else	
}		work_done = TRUE;	/*7*/
else {		}	
msgsend(src, 'done', size);	/*9*/		
active--;	/*10*/		
}			
}			
process_work();	/*11*/		

Figure 15. Example of a manager-worker communication scheme.

The workers conduct the following sequence. The worker sends a request for work to the manager (3) and receives a response (4). If the response is a partition of work (5), the worker does the work (6) and sends the result back to the manager (3). The message reporting the results to the manager also serves as a request for more work. If the response is a 'done' message (7), the worker exits.

In contrast to the Numerical Integration and Dirichlet Problems presented in Chapter 3, the manager code has no communication pattern to enforce synchronization of processes. The advantage of this scheme is that the work is distributed based on compute resources at run time. The amount of work in each partition may vary, and the manager does not know how long each partition will take to compute ahead of time. The wildcard receive operation allows the manager to assign work to worker processes *as they become available*.

4.2 The Challenge of Non-determinism in rMP Applications

Non-determinism presents challenges for the rMP technology. If process replicas have non-deterministic execution paths, consistent states cannot be guaranteed. Unfortunately, the assumption of consistent process replicas is fundamental to the resilience discussion to this point. At the lowest level, the multicast messaging protocols presume that messages should be consistent. Failure detection is based entirely on identifying anomalous behavior. More generally, resiliency is supported by redundant computation and regeneration of failed processes from any replica. These notions are preserved by the communication patterns in the integration and Dirichlet problem applications, but they do not hold in the presence of non-determinism in the manager-worker scheme.

The multicast protocols require careful construction to preserve the correctness of rMP applications using wildcard operations. Figure 16 displays the *msgrecv()* algorithm with the implementation of wildcard operations highlighted. Recall that if a specific source is designated in the *msgrecv()* call, messages are located by matching the resilient process id to the source of the message in the kernel queue. If ANY is designated, the module will receive the first message located in the queue for the process (5-6). Once the first message is located, the module replaces the source designated in the call with the source of the located message (9). The iteration continues to receive messages from the other members of the *updated* process group.

```
msgrecv(src,buf,size,&status)

msgs_recvd = 0;
while(msgs_recvd < r)  {                                    /*1*/
    for(i= 0; i<r; i++)  {                                  /*2*/
        res_id = resilient_id(src, i);                      /*3*/
        for_each_msg_in_queue  {                            /*4*/
            if((res_id == msg->src) || (res_id == ANY)){    /*5*/
                receive_and_compare_msg();                  /*6*/
                msgs_recvd++;                               /*7*/
                if(res_id == ANY)                           /*8*/
                    src = update_src(msg->src);             /*9*/
            }
        }
    }
    wait_for_messages();                                    /*10*/
    faiure_detection();                                     /*11*/
}
```

Figure 16. Pseudo-code for the *msgrecv()* call with wildcard operations.

The *msgrecv()* algorithm is conducted by replicas in a distributed fashion. However, the order of messages in the kernel queues may not be the same on different

hosts. As a result, when process replicas execute the same wildcard *msgrecv()* on different hosts, there is no way to guarantee that the modules will select messages from the same process group.

To better illustrate this problem, consider a simple manager-worker application running on two hosts, depicted in Figure 17. For simplicity, the application consists of a manager (M) and two workers (W1 and W2) with a level of resiliency of two (r = 2). The replicas in the figure are distinguished by the prime annotation (M and M').

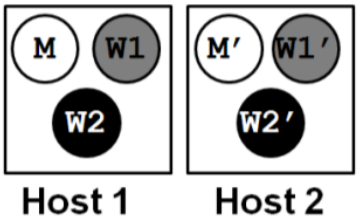


Figure 17. A simple manager-worker application with a manager (M) and two workers (W1 and W2) with level of resiliency of two (r = 2).

Figure 18 walks through several steps of the manager-worker algorithm to demonstrate the potential impact of wildcard receive operations on rMP applications. Each step is a snapshot of the application state, including the function that each process is executing and the contents of the process message queues at the beginning of the step. It is important to note that the figure depicts a particular instance of the application. However, because of non-determinism in the system, the computational progress of individual processes and the order in which messages are delivered are not guaranteed.

Step (1) begins after the initialization tasks of the algorithm. Each manager waits to receive work requests from ANY workers, and all workers send requests to the

managers. By step (2), the managers have messages in their queues. Each manager will receive the first message it locates. As a result, M receives a message from the W1 group, while M' receives a message from the W2 group. In step (3), M sends a response of the first partition of work (partition-1) to W1, while M' sends partition-1 to W2. In step (4), the manager continues to receive work requests. This time, M receives the request from W2, and M' receives the request from W1. In step (5), M sends partition-2 to W2, and M' sends partition-2 to W1.

Step (6) illustrates the first problem with the algorithm. The workers have inconsistent messages from M and M' in their queues. Message comparison would, therefore, indicate a process failure. If message comparison is omitted, the first message would be copied into the process address space, and the other discarded. Continuing to step (7), both W1 and W2 are computing partition-1. In step (8), all workers send the results of partition-1 to both the managers. In step (9), Both M and M' receive partition-1 results from W2 and send a response of partition-3 to W2 in step (10). Finally, in step (11), M and M' receive another copy of partition-1 results from W1. The results of partition-2 are never received, and the results processed by the managers will be *inaccurate*. This sequence of events demonstrates how independent wildcard receive operations undermine the correctness of rMP applications.

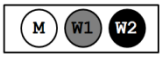
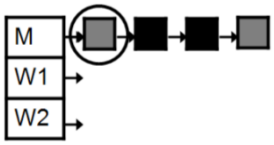
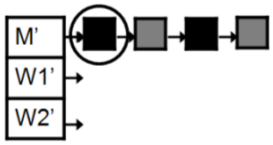
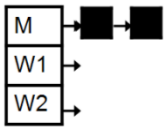
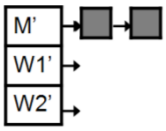
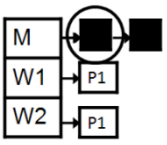
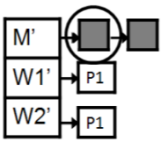
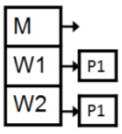
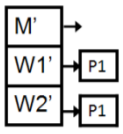
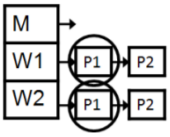
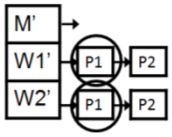
Step	Host 1 	Host 2 	Notes
1	M: recv(ANY, request) (4) W1: send(M, request) (3) W2: send(M, request) (3) 	M': recv(ANY, request) (4) W1': send(M, request) (3) W2': send(M, request) (3) 	
2	M: recv(ANY, request) (4) W1: recv(M, partition) (4) W2: recv(M, partition) (4) 	M': recv(ANY, request) (4) W1': recv(M, partition) (4) W2': recv(M, partition) (4) 	• The order of messages in the queue is non-deterministic. • M receives message from W1, but M' receives message from W2.
3	M: send(W1, P1) (7) W1: recv(M, partition) (4) W2: recv(M, partition) (4) 	M': send(W2, P1) (7) W1': recv(M, partition) (4) W2': recv(M, partition) (4) 	• M sends partition-1 to W1, but M' sends partition-1 to W2.
4	M: recv(ANY, request) (4) W1: recv(M, partition) (4) W2: recv(M, partition) (4) 	M': recv(ANY, request) (4) W1': recv(M, partition) (4) W2': recv(M, partition) (4) 	• Workers are still blocked because they are waiting for $r = 2$ messages.
5	M: send(W2, P2) (7) W1: recv(M, partition) (4) W2: recv(M, partition) (4) 	M': send(W1, P2) (7) W1': recv(M, partition) (4) W2': recv(M, partition) (4) 	• M sends partition-2 to W2, but M' sends partition-2 to W1.
6	M: recv(ANY, request) (4) W1: recv(M, partition) (4) W2: recv(M, partition) (4) 	M': recv(ANY, request) (4) W1': recv(M, partition) (4) W2': recv(M, partition) (4) 	• Workers receive inconsistent messages from M and M'. • Both W1 and W2 copy partition-1 into process buffer. Partition-2 is discarded.

Figure 18. Steps of a manager-worker communication scheme with resilient process groups. Process execution status and message queues are depicted for each step.

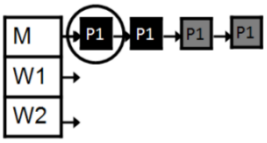
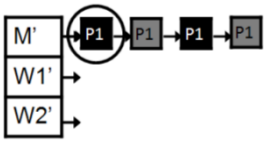
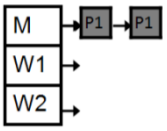
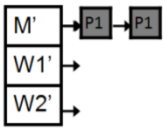
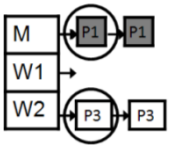
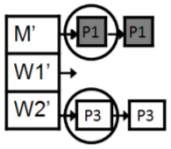
Step	Host 1 	Host 2 	Notes
7	M: recv(ANY, request) (4) W1: do_work(P1) (6) W2: do_work(P1) (6) 	M': recv(ANY, request) (4) W1': do_work(P1) (6) W2': do_work(P1) (6) 	• All workers are computing partition-1.
8	M: recv(ANY, request) (4) W1: send(M, P1) (3) W2: send(M, P1) (3) 	M': recv(ANY, request) (4) W1': send(M, P1) (3) W2': send(M, P1) (3) 	• Both workers report back results from partition-1.
9	M: recv(ANY, request) (4) W1: recv(M, partition) (4) W2: recv(M, partition) (4) 	M': recv(ANY, request) (4) W1': recv(M, partition) (4) W2': recv(M, partition) (4) 	• Both managers receive partition-1 results from queue.
10	M: send(W2, P3) (7) W1: recv(M, partition) (4) W2: recv(M, partition) (4) 	M': send(W2, P3) (7) W1': recv(M, partition) (4) W2': recv(M, partition) (4) 	• Managers send partition-3 to W2.
11	M: recv(ANY, request) (4) W1: recv(M, partition) (4) W2: recv(M, partition) (4) 	M': recv(ANY, request) (4) W1': recv(M, partition) (4) W2': recv(M, partition) (4) 	• Both managers receive partition-1 results again. • Results processed by the manager will be inaccurate.

Figure 18 (continued). Steps of a manager-worker communication scheme with resilient process groups. Process execution status and message queues are depicted for each step.

4.3 Proposed Solutions

The problem of non-determinism in systems that utilize replication is well known. Existing fault tolerant MPI libraries address the problem by disallowing wildcard receive operations [27] or by allowing process replicas to be inconsistent [25]. The rMPI library [26] provides an alternative solution by introducing consistency protocols between redundant processes. In these protocols, a primary process is chosen within a group of replicas, and it sends an extra message to the redundant processes when using wildcard operations to indicate the order in which messages are to be received. These protocols have been used in the context of static replication with a single set of redundant processes [26]. The rMP technology adapts this concept to arbitrary levels of resilience.

The adapted *msgrecv()* algorithm requires processes using the ANY parameter to coordinate with their process group. When the 0^{th} process replica in a group uses the wildcard receive operation, it first searches for a message in its queue. When a message is found, the communication module executes a coordination protocol in which messages are sent to the other replicas of the process group indicating the source and count of the message that were received. The module proceeds to receive messages from the remainder of the source process group to complete the *msgrecv()* call.

When a nonzero process replica calls *msgrecv()* with the ANY parameter designated, it waits to receive coordination messages from the 0^{th} replica. When a coordination message arrives, the module updates the source of the *msgrecv()* call to reflect the appropriate source, and the algorithm continues as if the ANY parameter was never used. As a result, the 0^{th} replica determines the execution path of all replicas in the

group. This restores the consistency of resilient processes using wildcard receive operations.

Unfortunately, the coordination protocols present a special case for resilience, due to the unique roll of the 0^{th} replica of a process group. Consider an instance in which a 0^{th} replica fails before a wildcard receive operation. The nonzero replicas will wait for coordination messages from the failed process before proceeding. As a result, the entire group will appear silent to other processes in the application. This behavior will not trigger regeneration because there is no anomaly within the group.

To solve this problem, a simple adaptive failure detection protocol is included within the coordination protocol. This protocol allows a nonzero process to trigger regeneration of the 0^{th} replica of its group. The triggering process must have messages in its *own* queue and wait longer than a constant timeout for a coordination message. This addition re-establishes failure detection for this special case. Figure 19 shows the modified pseudo-code of the *msgrecv()* algorithm with the coordination protocol highlighted.

```
msgrecv(src,buf,size,&status)

if((src == ANY) && (my_replica > 0))  {                    /*1*/
     wait_until_msg_are_in_queue();                        /*2*/
     wait_for_coordination_msg(src, timeout);              /*3*/
     if(timeout)                                           /*4*/
          trigger_failure();                               /*5*/
}

msgs_recvd = 0;
while(msgs_recvd < r)  {                                   /*6*/
     for(i= 0; i<r; i++)  {                                /*7*/
          res_id = resilient_id(src, i);                   /*8*/
          for_each_msg_in_queue  {                         /*9*/
               if((res_id == msg->src) || (res_id == ANY)){ /*10*/
                    receive_and_compare_msg();             /*11*/
                    msgs_recvd++;                          /*12*/
                    if(res_id == ANY) {                    /*13*/
                         src = update_src(msg->src);       /*14*/
                         send_coordination_msg(src);       /*15*/
                    }
               }
          }
     }
     wait_for_messages();                                  /*16*/
     faiure_detection();                                   /*17*/
}
```

Figure 19. Modified pseudo-code of the *msgrecv()* algorithm with the coordination protocol included.

4.4 Summary and Conclusions

Non-deterministic processes in applications utilizing replication undermine the fault tolerance of replication as well as the correctness of applications. A preliminary solution described here allows for the continued use of wildcard operations in the rMP technology, but it has several limitations. At a minimum, the addition of coordination protocols is inelegant. It complicates the *msgrecv()* algorithm by adding special cases.

With additions like these, it becomes more challenging to reason about execution paths within a program and to verify the correctness of the code.

There are also performance drawbacks associated with coordinating non-deterministic processes. The coordination protocols themselves incur some communication overhead. More than that, the approach compromises the intended function of the manager-worker communication scheme. The scheme is designed to dynamically allocate work to available processes. However, the proposed solution allocates work according to a single process of the group. If there are load disparities within the group, they may be exacerbated by this approach. In this instance, the solution serves to preserve functionality of the manager-worker scheme, but the performance benefit may be lost. This performance penalty is explored in Chapter 5 together with the other aspects of the technology.

Chapter 5 rMP Performance Benchmarks[3]

To evaluate the rMP technology, performance benchmarks were conducted using three distributed application exemplars. These benchmarks serve two purposes. The first is to evaluate the performance of failure-free execution of the rMP technology. The second is to demonstrate application resilience to process failures and to measure the performance impact of failure detection and process regeneration mechanisms. Throughout these benchmarks, the performance is discussed in terms of *time*. While other factors may contribute to application performance, such as power consumption, memory consumption, or computational load, they are not directly evaluated here.

5.1 Distributed Application Exemplars

Each exemplar is representative of a class of applications solved with a particular strategy: functional, domain, and irregular decomposition. Two exemplars were presented in Chapter 3 to demonstrate the rMP API. The Numerical Integration application represents problems solved with a functional decomposition strategy in which an algorithm is decomposed into components that are computed independently [68]. The Dirichlet exemplar represents problems solved with domain decomposition, in which the problem is decomposed over a large, static data structure [69].

To illustrate irregular decomposition, a LiDAR application is used, in which the data structures and algorithms evolve at run-time [71]-[74]. This real-world exemplar uses a sequential kd-tree algorithm to search LiDAR data to construct a digital terrain

[3] Significant portions of this chapter have been submitted for publication in the following:
- K. McGill and S. Taylor, "Operating System Support for Resilience," *IEEE Transactions on Reliability*, submitted for publication.

model. The LiDAR application uses a manager-worker messaging pattern, described in Chapter 4, with wildcard receive operations to delegate partitions of work to processes dynamically.

These applications are constructed as benchmarks using problem sizes that have a relatively short execution time, typically a few minutes. While this duration may not necessitate resilience, it is more than sufficient to benchmark performance and demonstrate the resiliency mechanisms.

5.2 Benchmark Platform

The benchmarks were executed on a dedicated Dell PowerEdge M600 Blade Server with 16 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 10.04.02 LTS Linux 2.6.32-32 x86_64. The system 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 with increasing levels of resiliency.

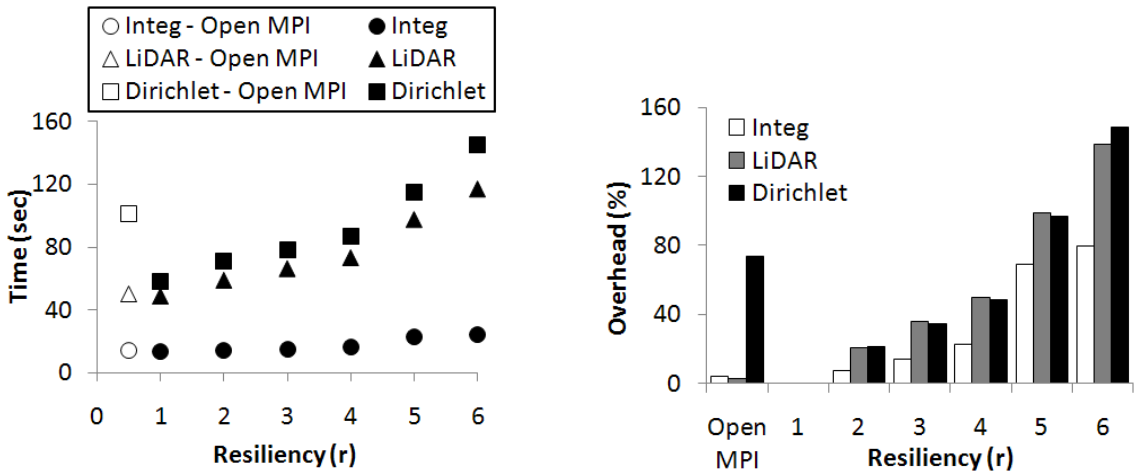
5.3 Failure-free rMP Performance

The performance of the rMP technology was evaluated to assess the impact of process replication and multicast messaging. For increasing levels of resiliency, the exemplars were executed ten times on the blade server, and the execution times were measured. The primary metrics used to assess performance are the average execution time of the application (i.e. wall clock time) and the overhead of resilience. The overhead of resilience is measured in terms of time—the percentage increase in average execution time of applications with replication. The performance was also compared to Open MPI to ensure that the rMP technology (without replication) does not incur

prohibitive overhead relative to message-passing libraries in the field. In the Open MPI tests, macros were used to convert between the rMP API and Open MPI library calls directly.

In these tests, a noticeable performance impact is expected due to both redundant computations and multicast communications of the rMP technology. Redundant computations are expected to increase the computational load of the application by a factor of r. However, because multicast messaging transforms a single application message into r^2 resilient messages, the communication overhead is expected to increase by a factor of r^2.

Figure 20(a) shows the average execution times of the exemplars using Open MPI and the rMP technology (r = 1, 2, 3, 4, 5, and 6). The overhead percentages are calculated with respect to the rMP applications without replication (r = 1) and are displayed in Figure 20(b). All three exemplars executed using rMP without replication outperform Open MPI. The average execution time of Open MPI is 4.2%, 2.9%, and 73.8% longer than the Numerical Integration, LiDAR, and Dirichlet applications executed with rMP without replication, respectively**.** This result demonstrates that the base rMP technology does not incur excessive overhead.



(a) (b)

Figure 20. (a) Average execution times and (b) overhead percentages of exemplar applications using Open MPI and rMP (r = 1, 2, 3, 4, 5, and 6).

The overhead of the rMP technology increases with the level of resiliency. To enable all mechanisms of rMP resilience, a minimum of triple resiliency is required. The overhead for rMP applications with triple resilience is 13.8%, 34.3%, and 35.6% for Numerical Integration, LiDAR, and the Dirichlet problem, respectively. This overhead is less than the factor of r overhead (300%) expected from redundant computations. However, these percentages can be explained, in part, by the computational load of the system as a whole. For $r < 4$, the blade server is under-utilized. There are compute resources available to absorb some of the overhead of redundant computations. This fact suggests that the overhead observed for applications with $r < 4$ may have other causes, such as synchronization within the multicast protocols or, in the case of the LiDAR exemplar, overhead in the coordination protocols.

Figure 20 shows a slight elbow in the plots at $r = 4$. This elbow corresponds to the capacity of the blade server. For $r > 4$, the system is over-utilized, and processes are directly competing for resources. Figure 21 shows the *full load overhead* of rMP applications with $r \geq 4$. The full load overhead is the percentage increase in average execution time of the exemplars *with respect to the r = 4 average execution time.* This metric allows assessment of the over-utilized system under. The expected overhead is included as a percentage increase in the number of processes with increasing levels of resiliency (i.e. applications with $r = 5$ have 25% more processes than applications with $r = 4$). Figure 21 demonstrates that the full load overhead is comparable to the expected overhead from redundant computation for all three exemplars. With further exploration, these benchmarks can be used to model the overhead of the rMP technology due to redundant computations and communications. However, other factors, such as the operating system scheduling policies and the multicast synchronization complicate this analysis.

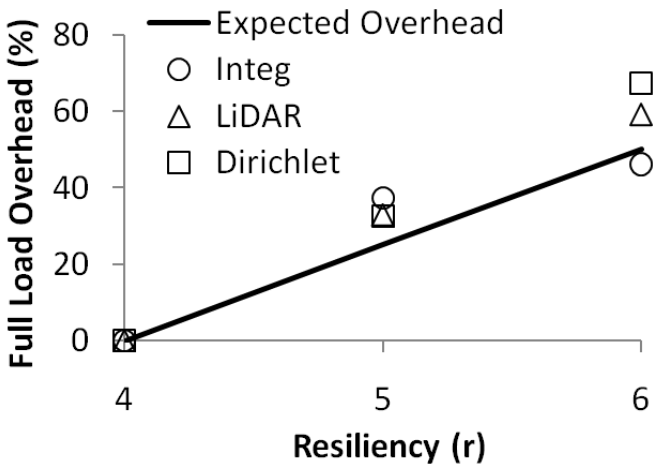


Figure 21. Overhead of rMP applications with respect to rMP with $r = 4$. The expected overhead is calculated as the percentage increase in the number of processes with respected to $r = 4$ applications.

5.4 rMP Performance with Process Failures

To demonstrate resilience, the exemplars were executed with process failures. Two sets of experiments were conducted. One set included *complete* process failures. In each execution of the exemplars, one process, selected at random, was manually killed. The second set of experiments included *Byzantine* failures, or compromised processes. Process compromise was emulated by copying a random byte of data into the buffer of an outgoing message. In each execution of these experiments, one process, selected at random, was compromised in this way.

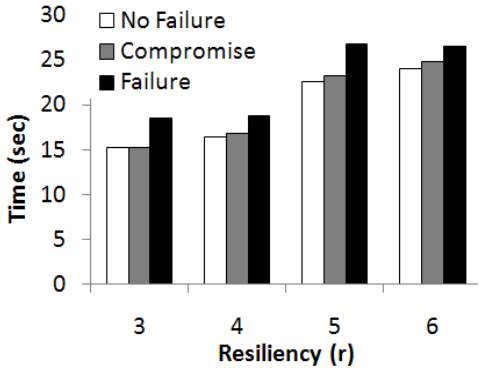
The metrics associated with rMP performance with process failures include the overhead of regeneration, the averaged failure detection times, and the average duration of process regeneration. Like the previous overhead, the overhead of regeneration is measured in terms of time, as the difference in average execution time of applications

with and without regeneration. The failure detection times are measured from the time at which the failure is injected until it is detected. The duration of process regeneration is measured from when the process replica is interrupted until the regenerated process resumes execution on its new host.

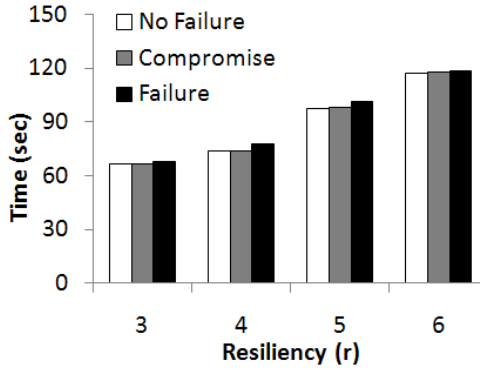
In these experiments, only one process out of 92 – 192 total processes of the application is killed. As a result, the overhead is not expected to be a significant percentage of the total execution time. The overhead is associated with a single failure/detection/regeneration event. This is the reason for calculating the overhead of regeneration as a difference in execution time rather than a percentage increase. This event-based overhead is expected to be independent of the total execution time of the application.

5.4.1 Overhead of Regeneration

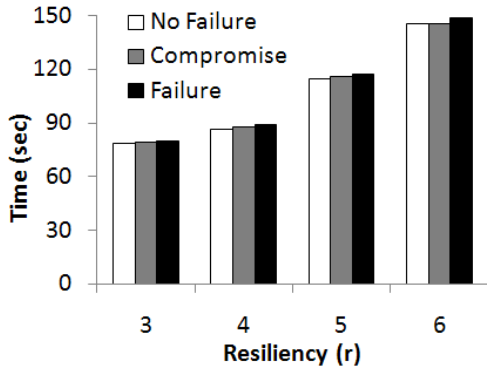
Figure 22 shows the average execution times of both sets of rMP experiments with process failures for the Integration, LiDAR, and Dirichlet exemplars. The average execution times of the exemplars without failure are included for reference. In all three exemplars, the average execution time is greater for the experimental set that includes failed processes than for the set that includes compromised processes. As expected, it appears that the increase in execution time for process regeneration is a small percentage of the total execution time of resilient applications. The Numerical Integration exemplar may be an exception to this generalization because of its relatively short execution time.



(a)



(b)



(c)

Figure 22. Average execution times of the (a) Integration, (b) LiDAR, and (c) Dirichlet exemplars without failure, with a single compromised process, and with a single failed process for $r = 3, 4, 5$, and 6.

The overhead due process regeneration for all three exemplars is shown as a function of resiliency in Figure 23. This overhead is attributed to both the failure detection times and the duration of process regeneration. The overhead in the set with compromised processes is one second or less for all exemplars, while the overhead in the set with complete failures ranges from 1.3 to 4.3 seconds. This disparity in overhead implies that the time to detect complete failures through communication timeouts is the predominant source of overhead. There is no clear trend in the overhead as a function of resiliency.

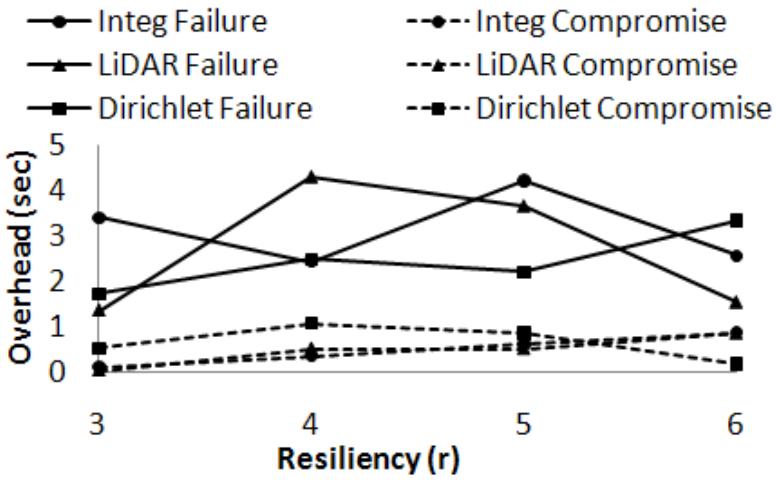
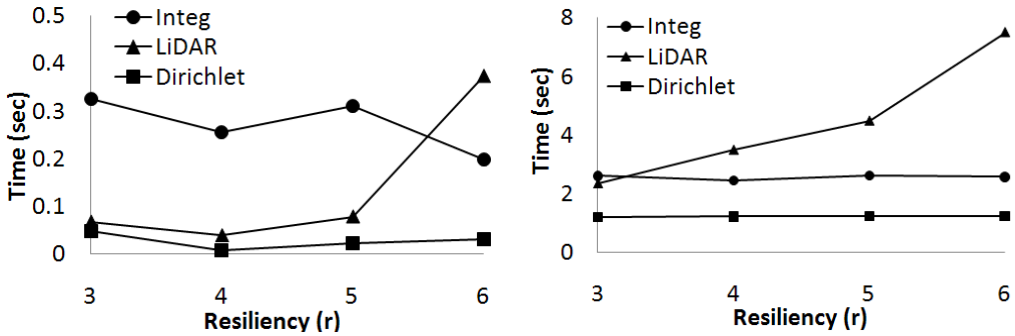


Figure 23. The overhead of a single compromised process and a single failed process for r = 3, 4, 5, and 6 of all three exemplars.

In both sets of experiments, the overhead incurred due to process regeneration is less than the systematic overhead of resilience displayed in Figure 20. This result suggests that there is negligible performance impact to supplementing the base rMP technology with failure detection and dynamic process regeneration mechanisms.

5.4.2 Failure Detection Time

Figure 24 shows (a) the average time to detect compromised processes through message comparison and (b) the average time to detect failed processes through adaptive communication timeouts for all three exemplars. The average times to detect failures through message comparison range from 0.01 to 0.4 seconds. The average times to detect failures through communication timeouts range from 1.2 to 7.5 seconds. This disparity in detection time between process failure modes is consistent with the observed overhead of regeneration.



(a) (b)

Figure 24. Average time to detect (a) compromised processes through message comparison and (b) failed processes through adaptive failure detection as a function of resiliency for all three exemplars.

The failure detection times of the Integration and Dirichlet exemplars are not correlated with the level of resiliency. In fact, the adaptive failure detection times are relatively constant for increasing resiliency. However, the detection times increase with resiliency for the LiDAR exemplar. This result reveals the sensitivity of the adaptive

failure detection algorithm to application granularity. The LiDAR exemplar is the most coarse-grained application. This leads to a larger computational term in the adaptive timeout calculation, and this effect is expected to increase with the level of resiliency.

The LiDAR exemplar's use of wildcard receive operations provides another explanation for the increase in detection time. Recall the speculation in Chapter 4 that the coordination protocols may cause the computational load to be distributed unevenly across the hosts. This phenomenon would lead to larger variations in the compute time within a process group. Again, this effect is likely exacerbated at increased levels of resiliency. Without the computational term of the timeout, the LiDAR exemplar triggers false positives in the failure detector. This observation reinforces that the increase in detection time is a product of increasing variations in message latency for LiDAR processes.

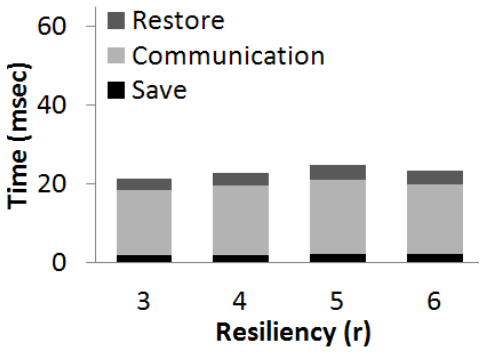
5.4.3 Process Regeneration Duration

Figure 25 displays a breakdown of the average times to perform process regeneration per MB of the process image size for all three exemplars. The durations have been normalized by the image size because processes are randomly selected for failure and compromise, and the size of the image directly impacts the amount of work performed in regeneration. Table 1 provides the approximate image sizes for regenerated processes of each exemplar. Recall that the LiDAR exemplar uses an irregular decomposition strategy in which the work is partitioned at run time. As a result, the images of LiDAR processes that are regenerated range from 3MB to 55MB, compared to 1MB and 9MB of the Integration and Dirichlet exemplars, respectively. The process regeneration is broken down into three subtasks: save, communication, and restore. The

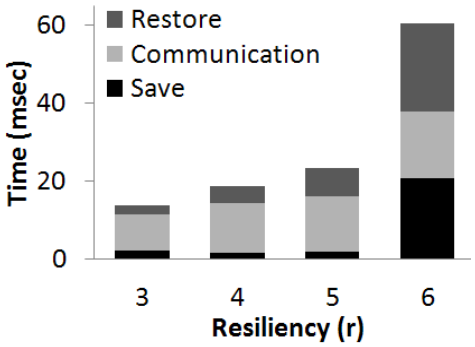
communication subtask includes the communication module protocols, such as sending update messages, forwarding application messages, and sending the process image to the destination host.

Table 1. Distributed Application Exemplar Image Sizes

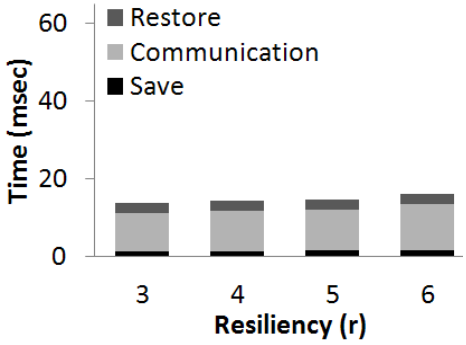
Exemplar	**Approximate Image Size (MB)**
Numerical Integration	1
Dirichlet Problem	9
LiDAR	3-55



(a)



(b)



(c)

Figure 25. Average times to perform process regeneration normalized per MB of process image size for the (a) Integration, (b) LiDAR, (c) and Dirichlet exemplars.

The total regeneration duration for all exemplars ranges from 14 to 60 msec per MB. For all the Integration and Dirichlet exemplars, these values result in less than 150

msec for a single process regeneration. The majority of the regeneration time is spent during the communication subtask, and most of this time is spent sending the process image to the destination host. The Integration and Dirichlet exemplars show no significant trend in the duration of regeneration with the level of resiliency.

The regeneration times for the LiDAR exemplar increase with the level of resiliency. In particular, the average times to save and restore the process image increase with resiliency. The relatively large image size of select LiDAR processes may be the cause of this increase. The operation of saving and copying very large memory areas in kernel space incurs disproportionate overhead. The manual operations of walking a process address space from out of context and swapping pages into memory may become more significant for large images. As the level of resiliency increases, there are more processes executing on a single machine, and the likelihood that pages have been swapped out of memory increases. These effects and others may complicate the analysis of regeneration for process images that exceed 10MB.

5.5 Summary and Conclusions

These benchmarks serve to gauge the performance of the rMP prototype and provide an input for future development. For some rMP applications, triple resiliency incurs up to 35% overhead relative to Open MPI. This performance impact may not be tolerable by all mission-critical applications that require resilience. However, the current rMP prototype is largely un-optimized. There are opportunities for performance enhancement that would not compromise resilience. For example, all incoming messages are placed in the kernel message queues, even if the destination process is already waiting for the message. A more advanced mechanism would place the message directly in the

user-space buffer of the destination process and eliminate a memory operation. This example is representative of the potential for optimization in the rMP technology.

The overhead of process regeneration is relatively minor compared to the systemic overhead of the rMP technology. This fact suggests that the benefit of resilience over static replication outweighs the modest addition of overhead. Moreover, the overhead due to process regeneration is only incurred when process failures are detected. This is a distinct advantage of rMP over checkpoint/restart technologies, in which periodic checkpoints are required regardless of the failure history.

The largest source of overhead for process regeneration is the adaptive failure detection algorithm. To date, several algorithms with variable communication, computational, and constant components have been explored. These benchmarks suggest that some application characteristics disproportionately affect the performance of failure detection, such as the granularity or the use of wildcard operations. It is likely that an increase in the diversity and scale of applications utilizing the proposed algorithm would reveal more variations in the observed performance. For example, imagine a system in which message latencies are characterized by a bimodal distribution. In this instance, it is unlikely that an algorithm based on average latencies would be a reliable indicator of process failure. More advanced algorithms that incorporate statistical observations, such as distribution densities of message latencies, may be beneficial. Further exploration of alternative failure detection algorithms is warranted to better understand these potential effects.

These benchmarks have been conducted on a dedicated platform of 16 hosts connected by a 1 Gbps Ethernet network. While this platform provides a controlled

environment for benchmarking, it is not necessarily representative of real-life networks in which large fluctuations in network and computational loads may occur. The impact of system utilization on the overhead of resilience has already been examined, but these more strained environments may have other effects on the rMP technology, such as increases in failure detection times and the duration of process regeneration. In addition, the failure detection algorithms may incur false positives by mistaking delayed processes, executing on over-utilized hosts, for failed processes. These events were observed in development during which hosts of the platform were used for other tasks. Significant variations in the computational load of hosts on the network caused a number of false positive failure detections and unnecessary process regenerations. However, these applications still maintained resilience and operated through the perceived failures. These observations motivate further performance evaluation and optimization of the rMP technology on field systems that better represent realistic environments.

Chapter 6 Survey of Resilient Process Management Strategies[4]

Recall that in Chapter 3, a simple mapping scheme was presented based on modular arithmetic. To support the rMP mechanisms in general operating systems, distributed process scheduling algorithms are required that include the notions of replicated processes and process group locality. These requirements are in addition to the traditional priorities of balancing the system load. This thesis explores the challenge of resilient process management from two different perspectives. Traditional diffusive algorithms provide a generic, automatic, and distributed process scheduling algorithm for the purposes of load-balancing. Robotic swarming algorithms achieve a variety of goals while imposing swarm cohesion, or locality between swarm members. The swarming perspective represents a shift in process scheduling from a single algorithm exclusive to the operating system toward cooperation between processes and the operating system to allow processes to *manage themselves*. This work combines concepts from both perspectives to integrate the goals of resilience and performance in a single strategy.

6.1 Distributed Resource Management

Distributed resource management is a well-established field. Diffusive algorithms [75]-[81], dimension exchange methods [82]-[84], and gradient models [85]-[86] are among the most prevalent approaches. Of particular interest is a parabolic diffusion scheme based on the heat diffusion equation, proposed by Heirich and Taylor [76]. This model characterizes process load as a scalar *heat* value and automatically disperses

[4] Significant portions of this chapter have been published previously in the following:
- K. McGill and S. Taylor, "Robot Algorithms for Localization of Multiple Emission Sources," *ACM Computing Surveys (CSUR)*, vol. 43, no. 3, 2011.

processes in the architecture. This approach has several attractive properties: It is a simple, scalable algorithm that uses a completely local iteration and only nearest neighbor communication. It provides global convergence to a balanced CPU load and guarantees for global convergence and progress through well-established mathematical analysis. The algorithm has been shown, through simulation, to simultaneously balance multiple independent load distributions over large-scale architectures representing concurrent computations injected at random locations with disparate random loads [76]. Extensions to the algorithm allow multiple properties, including communication, memory, and CPU load, to be balanced simultaneously [81].

6.2 Robotic Swarming Algorithms

Robotic swarming algorithms present an alternative view of resource management in which swarms of distributed and autonomous agents achieve a common goal while imposing emergent properties of *swarm cohesion* and *obstacle avoidance*. There is a large field of research that utilizes robotic swarms for emission source localization; a survey of this research is presented in a prior publication [87]. In emission source localization, robots sample the *gradient* of some physical or chemical property in order to locate potential sources. A large number of robots, equipped with sensors and inter-robot communication, may collectively search for all sources in minimal time. This problem is analogous to the load-balancing problem in that a large number of processes sample load and search for troughs in processor utilization. As a result, a direct application of robotic swarming algorithms is found in distributed process management. In addition, this application allows emergent swarm properties to be leveraged for the requirements of resilience.

There are several classes of algorithms in the robotics literature that are candidates for process scheduling: 1) biologically-inspired approaches based on *E. Coli* bacteria [88], glowworms [89]-[92], and other social foraging organisms [93]-[95]; 2) population- and evolutionary-based models, including genetic algorithms [96] and Particle Swarm Optimization [97]-[98]; and 3) probabilistic models based on Bayesian occupancy grid mapping [99].

The application of robotic swarming algorithms to resource management is relatively new [100]-[103]. In general, these efforts capitalize on the ability of distributed processes to use collective swarm intelligence to balance load. However, the swarming algorithms that have penetrated the space have excluded key properties required by resilience.

Three properties of swarming algorithms are most relevant to the process management problem: how to partition or spread the swarm across the architecture for system-wide utilization, how to maintain locality within process groups, and how to avoid collisions with other processes (replicas) and architecture boundaries during the search for resources. Several candidate process scheduling algorithms from the robotics literature are presented with an analysis of their potential contributions to these challenges in process management.

6.2.1 Biased Random Walk (BRW)

The BRW algorithm [88], shown in Figure 26, is inspired by the chemotaxis of *E. Coli* bacteria. All processes act independently without communication or synchronization and execute two actions: a *run* and a *tumble*. A run represents a process movement in a straight line, and a tumble is a random reorientation in a new direction. The presence of a

load gradient affects the length of the BRW process's run. If a run is in the direction of a positive gradient, the length of the run is extended by a bias step. By extending the length of runs in the direction of the positive gradient, the process gradually moves toward resources. The BRW pseudo-code in Figure 26 has a step length of ten units and a 10% bias. Processes conducting a BRW have no prior knowledge of the architecture.

```
bias = 10%;
step_length = 10;
step_extension = step_length*bias;
C1 = measure_local_resources();
while (target_not_found)  {
     direction = tumble();
     run(direction, step_length);
     C2= measure_local_resources();
     if( C2 > C1)  {
          run(direction, step_extension);
          C2= measure_local_resources();
     }
     C1 = C2;
}
```

Figure 26. BRW pseudo-code for run and tumble sequence with 10% bias [88].

This BRW algorithm has been simulated in the robotics literature on 100 agents in the presence of constant and time varying gradient sources, modeled by inverse square law, cubic, linear, and exponential profiles. When agents are deployed randomly in the presence of multiple gradient sources, the authors observe that all sources are located, and sources with higher intensities are tracked by a larger fraction of agents. Further, the attraction of agents to sources is dynamic, as agents adapt to the addition and removal of gradient sources from the space.

One property of the process management problem is addressed by the BRW algorithm: there is an inherently random dispersion of the swarm in which processes are diverted to different resources in the architecture. The other process management concerns are disregarded: process collisions are ignored and there is no communication between processes to enforce locality.

6.2.2 Glowworm Swarm Optimization (GSO)

The Glowworm Swarm Optimization (GSO) algorithm [89]-[92] models processes on *glowworm* behavior. Each GSO process possesses a luminescence quantity called *luciferin* that causes them to glow in response to the local field, corresponding to the amount of CPU resources available. In nature, female worms glow to attract mates, but, in the GSO algorithm, the glow attracts other processes to engage in a cooperative search. Each process has a limited communication range and an adaptive decision range that is less than or equal the process's communication range. A process selects neighbors that are within its decision range and have higher luciferin values.

To begin a search for resources, a process chooses a leader from its neighbors and moves toward it. The most probable choice for the leader is the neighbor with the highest luciferin value, corresponding to the likely direction of a load trough, or a cluster of under-utilized computers. As a result of this leader selection, subgroups form within the swarm and begin searching for nearby load troughs. Figure 3 provides a reproduction of the GSO algorithm [90].

deploy_agents_randomly;
For all i, set $l_i(0) = l_0$ //Set initial luciferin level of glowworm i
For all i, set $r^i_d(0) = r_0$ //Set initial local decision range of glowworm i
{
for each glowworm i do:
//Update the luciferin level at current time
$l_i(t) \leftarrow (1-\rho)l_i(t-1) + \gamma J(x_i(t));$ (1)

for each glowworm i do {
//Determine neighbors of glowworm *i*
//in the local-decision range and with higher luceferin levels
$N_i(t) = \{j: d_{ij}(t) < r^i_d(t);\ l_i(t) < l_j(t)\};$ (2)

//For each neighbor
For each glowworm $j \in N_i(t)$ do:

//Calculate probability of selecting neighbor *j*

$$p_j(t) = \frac{l_j(t) - l_i(t)}{\sum_{k \in N_i(t)} l_k(t) - l_i(t)} \quad (3)$$

//Select neighbor
j = select_glowworm_j; // (using $p_j(t)$)

//Move toward neighbor *j*

$$x_i(t+1) \leftarrow x_i(t) + s\left(\frac{x_j(t) - x_i(t)}{\left\|x_j(t) - x_i(t)\right\|}\right); \quad (4)$$

//Update local-decision range based on specified number of neighbors
$r^i_d(t+1) \leftarrow min\{r_c, max\{0, r^i_d(t) + \beta(\eta_t - |N_i(t)|)\}\};$ (5)

} $t \leftarrow t + 1;$
}

Figure 27. Glowworm Swarm Optimization (GSO) algorithm [90].

Initially, all glowworms have constant luciferin ℓ_0 and a constant local decision range r_0. The position of glowworm *i* at time *t* is $x_i(t)$. Each glowworm's luciferin level $\ell_i(t)$, depends on the luciferin decay constant ρ, the luciferin enhancement constant γ, and

the concentration at process i's location, $J(x_i(t))$. Glowworm i has a neighborhood $N_i(t)$. Glowworm j is in $N_i(t)$ at time t if the distance $d_{ij}(t)$ between glowworms i and j is within the local decision range $r^i_d(t)$ and the luciferin level of process j is higher than process i. At time t, each glowworm selects a leader from $N_i(t)$ with probability $p_j(t)$. The probability $p_j(t)$ is the ratio of the excess luciferin of process j, $\ell_j(t)$- $\ell_i(t)$, to the total excess in luciferin of glowworms in $N_i(t)$. Glowworm i updates its position by moving one step s toward glowworm j. The direction to glowworm j is calculated from the normalized vector from position i to j, $x_j(t)$- $x_i(t)$. After each time step, the local decision range is updated depending on the communication range r_c, a predetermined number of neighbors η_t, and a constant β.

The algorithm has been simulated in an extensive set of multimodal environments with up to one hundred gradient sources. Simulations show that the method succeeds in locating all gradient sources under *some* initial conditions. Success is not guaranteed on all initial configurations, but the algorithm shows promise as a viable general solution.

The GSO's adaptive decision range is a vital feature of the algorithm. It provides the ability to partition the swarm into subgroups, without global communication, to search for multiple load troughs simultaneously. This feature is ideal for the load-balancing problem in which the goal is to utilize the entire system. In addition, because the adaptive decision range is limited by the communication range, processes maintain locality with these subgroups. GSO processes are never attracted away from their group. Unfortunately, the authors incorporated a low level sensory-based obstacle avoidance model for the robotics realm, but it is not applicable to process management.

6.2.3 Biasing Expansion Swarm Approach (BESA)

The Biasing Expansion Swarm Approach (BESA) is designed to locate an unspecified number of gradient sources in a large, unknown domain [93]. BESA processes obey the three basic swarm control rules of *separation*, *cohesion*, and *alignment* with a bias toward areas of higher resource concentration. Central to the BESA algorithm is the formation of a global ad-hoc network in which processes are limited to local communication. Data packets can be forwarded throughout the swarm via multi-hop paths that connect processes that are otherwise out of communication range. Through this ad-hoc network, each process can incorporate out-of-range process locations and load statistics into its local BESA algorithm.

Each process in the swarm represents the domain as an occupancy grid map. The local communication range is limited to adjacent cells in the grid. After initial distribution and the establishment of an ad-hoc network, processes share their locations and local load statistics with the swarm. Each process then incorporates this information into its own occupancy grid. Figure 28 is a graphical representation of a process's occupancy grid map taken from [93]. Each cell is identified as an explored cell, an occupied cell, an un-explored cell, or an un-explored expansion cell.

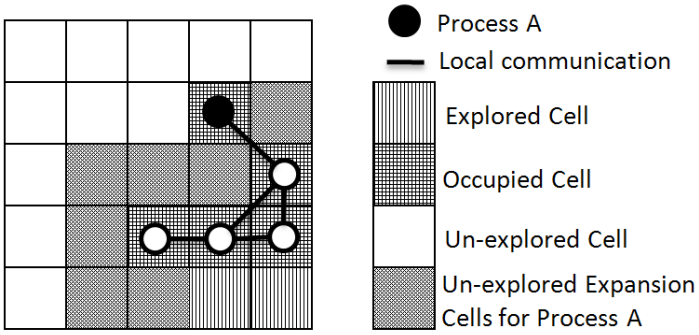


Figure 28. The domain from Process A's perspective [93].

Each process uses the occupancy grid to plan its next target cell within the framework of the swarm rules. To uphold the rule of *separation*, a process avoids entering a cell already occupied by another process. To maintain swarm cohesion, the process uses a gradual expansion algorithm [94]. This algorithm restricts the process's target cell to expansion cells, defined as cells that are unexplored, unoccupied, and adjacent to at least one other process in the swarm. Figure 28 shows the expansion cells for an individual process. This gradual expansion algorithm maintains the cohesion property of the swarm as well as the ad-hoc network connectivity.

In order to steer the swarm toward the available resources, the BESA algorithm assigns a biasing parameter to each of a process's expansion cells. The biasing parameter, *B(x, y)* is determined by the following equation:

$$B(x,y) = \frac{K}{n} \times \sum_{i=0}^{n} \frac{C_i}{r_i^2} \qquad (3)$$

In (3), n is the total number of processes that communicate their local load statics, C_i is the local load of process i, K is a constant, and r_i is the distance between expansion cell *(x,y)* and the cell in which process i is located. The process will choose the expansion cell with the highest biasing parameter as its target cell. The effect of this

biasing equation is that processes target empty cells that are nearest to the cells where the greatest amount of available resources are measured. As a result, the swarm as a whole moves in the direction of available resources.

The authors have simulated the BESA algorithm on a swarm of twenty agents in the presence of two gradient sources modeled on an inverse square law distribution. The performance of the method was compared to a gradient seeking approach. Under a variety of conditions, the BESA algorithm converges on all sources in approximately half the time of the gradient seeking algorithm.

The BESA algorithm clearly addresses two concerns of the process management problem: maintaining locality with other processes of the swarm and avoiding process replicas during the search. Both properties are achieved through the gradual expansion algorithm. Process movements are limited to cells of the occupancy grid that are unoccupied and adjacent to another process of the swarm. Unfortunately, because the occupancy grid constructs a relatively rigid definition of the space, the gradual expansion algorithm prevents the swarm from partitioning or expanding in the domain.

6.2.4 Particle Swarm Optimization (PSO)

Particle Swarm Optimization is a population-based model introduced by Kennedy and Eberhart [104] in which a swarm of particles, or processes, move in a virtual space to find an optimal solution. PSO is similar to evolutionary-based algorithms in that the swarm represents a population of solutions that are characterized by their fitness. These algorithms have been used to locate multiple gradient sources by treating concentration measurements as fitness. In the context of process management, the local CPU load serves as the fitness measurement. PSO assumes a process can communicate with

neighbors and evaluate a value of its fitness at each location. The motion of each process is governed by two equations:

$$\upsilon_i(t) = \phi v_i(t-1) + \rho_1(x_{pbesti} - x_i(t-1)) + \rho_2(x_{gbest} - x_i(t-1)) \quad (4)$$

$$x_i(t) = x_i(t-1) + \upsilon_i(t) \quad (5)$$

In equations (4) and (5), $v_i(t)$ and $x_i(t)$ are the velocity and position vectors of process i at time t, respectively. The quantity $x_{pbest\ i}$ is the previous position at which process i had the best value of fitness, and x_{gbest} is the previous position of the best value of any neighbor of process i. Vectors ρ_1 and ρ_2 contain random positive values for each dimension of the state space, and ϕ is a constant that controls the magnitude of velocity. These equations steer a process toward a target by directing motion to the best previous positions of the swarm.

A distributed PSO algorithm has been proposed that explicitly aims to locate multiple gradient sources when swarm agents are deployed in a corner of the domain [97]. The search for resources is divided into two stages: global search and local search. A global search is conducted in the absence of load information and encourages exploration of the domain. During a global search, a potential field-based method is used to control the motion of swarm processes. Processes are repulsed from obstacles and other processes and move to create separation from them. Once available resources are encountered, a local search is initiated using the PSO algorithm (see Figure 29).

```
initialize_population();

While(targets_not_found) {
    evaluate_individual_fitness();
    compare_neighbors_fitness();
    calculate_target_velocity();
    calculate_target_position();
    move();
}
```

Figure 29. PSO-based search algorithm pseudo-code [97].

The algorithm was simulated with 10 agents and 5 gradient sources in three environments of varying stability. To evaluate the performance of the PSO algorithm, a bacterial chemotaxis algorithm and a generic gradient ascent algorithm were included in simulations. Generally, the PSO-based algorithm performed comparably to the other two algorithms, but its relative performance increased as the gradient profile became more irregular.

This PSO algorithm makes two contributions to the process management problem, but both are included in the global search component of the algorithm. The algorithm treats other processes and obstacles as repulsive forces on a process. This enables process to spread and explore the domain for resources. In addition, because the processes know the positions of the other processes from local communication, process replicas are avoided before contact.

6.2.5 Survey Summary

Swarming algorithms have evolved partial solutions to the process management problem, but they are not integrated in a cohesive manner. Several algorithms provide

one or two desirable properties for process management but not all of them in a single strategy.

In the context of partitioning or spreading processes of the swarm across the architecture to locate all resources, the most advanced work is evident in the GSO technique. The adaptive local-decision domain enables clusters to form within the swarm and search for nearby resources [90]. The BRW algorithm utilizes random dispersion because the processes are conducting independent randomized searches but does not enforce this property [88].

Two algorithms explicitly maintain locality between processes. GSO processes are attracted to neighbors within their local communication range with greater resources available [90]. Because this attraction is the only driving force for process mobility, processes maintain locality with their group. The BESA algorithm incorporates locality in the gradual expansion algorithm [93]-[94]. This algorithm restricts the process movement to maintain adjacency with at least one other process in the swarm.

Tactics have also been recommended to avoid other process replicas and boundaries during the search. The PSO algorithm treats neighbors and boundaries as repulsive objects, providing the ability to separate replicas [97]. The BESA algorithm overtly prevents processes from moving into occupied cells of the architecture map [93]-[94].

Finally, the literature includes several independent algorithms that are conceived, implemented, and analyzed either in isolation, or with reference to another, often poorly documented, algorithm [88], [90], [93], [98], [99]. The lack of a common set of validation cases and reference algorithms that form ground truth for comparative analysis

makes it impossible to directly compare the different algorithms and weigh their merits for different applications. This begs the question of what should such a reference set involve and how should comparative analysis be performed. Lastly, there are the questions of assessing convergence and sensitivity analysis. Many of the algorithms involve randomization as a primary ingredient of their approach. To ensure that results are meaningful, it is valuable to establish a common basis that allows not only the average convergence to all resources to be reported but also an error bar established through a minimum number of simulation runs.

Chapter 7 Establishing Ground-truth in Process Management[5]

In order to unify the field for development, a common set of validation benchmarks were proposed that provide *ground-truth* for comparative analysis of swarming algorithms for process management [34], [35]. The benchmarks capture the primary first-order attributes of the general process scheduling problem. They are used, in combination with sensitivity analysis, to conduct a comparative analysis of existing approaches and to provide meaningful insights into the relative load-balancing performance of algorithms.

Two algorithms from the robotics literature have been selected as candidates for this study: BRW [88] and GSO [89]-[92]. Both algorithms supply the means for swarms to partition or spread throughout the architecture and search for available resources. In addition, they have successfully located multiple disparate gradient sources in a variety of simulations [18], [89]. The BRW stands out as particularly valuable for comparative analysis because it enables autonomous process mobility without communication and is simple to implement. The GSO algorithm is attractive because of its ability to maintain locality through limited range communication. For these reasons, these algorithms are explored in the initial ground-truth analysis.

---

[5] Significant portions of this chapter have been published previously in the following:

- K. McGill and S. Taylor, "Comparing swarm algorithms for large scale multi-source localization," *In Proc. of the IEEE Conf. on Technolologies for practical Robot Applications*, Woburn, Massachusetts, Nov 2009.
- K. McGill and S. Taylor, "Comparing swarm algorithms for multi-source localization," *Proc. of IEEE Int. Wkshp. on Safety, Security, and Rescue Robotics*, Denver, Colorado, Nov 2009.

7.1 Benchmark Cases

The standard set of benchmark cases used to compare algorithms is presented in Figure 30 [34]. These benchmarks represent a two-dimensional mesh architecture with variable resources available on each CPU. Lighter areas of the figure indicate a greater amount of resources available. The same static load distribution is included with three alternative initial process distributions. The pronounced light peaks in the benchmark are load troughs, or clusters of CPUs with available resources. The load troughs vary in size (width) and intensity (amount of resources available). The characterization and distribution of the load troughs ensure that troughs are occluded by other troughs of lesser, greater, or equal intensity. Extensive dead space, representing fully loaded computers, is included to determine the impact of an imperceptible gradient in the amount of CPU resources available on load-balancing performance. Although a two-dimensional mesh is presented, the problem can be applied to any architecture through an appropriate virtual to physical machine mapping.

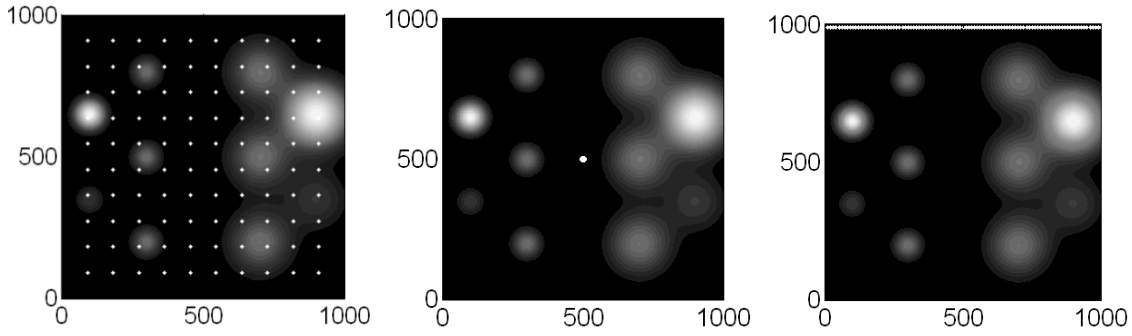
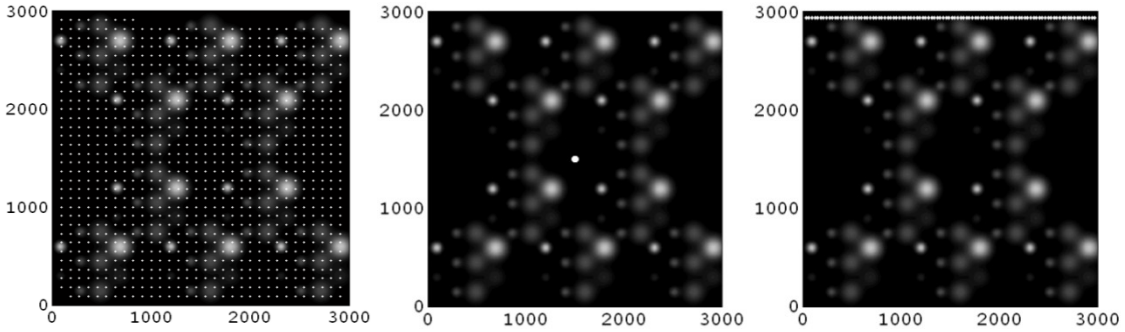


Figure 30. Standard benchmark cases: (a) uniform initial distribution, (b) point initial distribution, and (c) line initial distribution.

All three benchmarks in Figure 30 contain the same gradient field with ten Gaussian load troughs on a 1000 x 1000 unit mesh (1,000,000 CPUs). The load troughs are 5 to 25 units in height and approximately 100 to 300 CPUs in width. Dead space is created by setting the resource value to zero if it is below a threshold value of 0.5. Three initial process distributions are the *uniform*, *point*, and *line* distributions. The uniform distribution, common in the swarming literature [88], provides total coverage of the mesh and might correspond to a uniformly scattering process scheduler. The point distribution represents initial process distribution on a small collection of CPUs in the center of the mesh. The line distribution corresponds to mapping processes along one dimension of the architecture. Appendix 1 includes a complete definition of the standard benchmark cases for repeatable experiments.

A large scale benchmark set was devised to assess the scalability of competing algorithms. Figure 31 illustrates the large scale benchmark set [35]. All three benchmarks contain 100 Gaussian load troughs on a 3000 x 3000 unit mesh (9,000,000 CPUs). The field is constructed by replication of a parameterized group of 10 troughs. The same conventions are used to create dead space, and the same initial process distribution strategies are employed. Details of the large scale benchmarks are presented in Appendix 2.



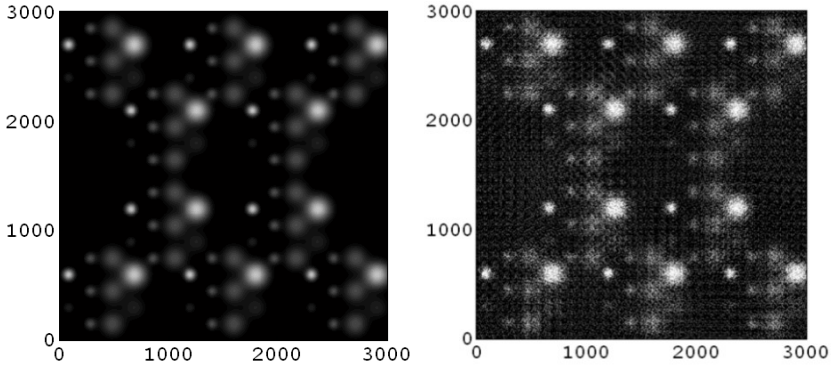
(a) (b) (c)

Figure 31. Large scale benchmark cases: (a) uniform initial distribution, (b) p initial distribution, and (c) line initial distribution.

The benchmarks in Figure 30 and Figure 31 feature a smoothly-varying gradient field to serve as an initial test case. However, the nature of the gradient source model dictates the gradient profile and may impact load-balancing performance. For example, if the load troughs were modeled to obey an inverse square law or step function, the relative algorithm performance might be altered. For these reasons, it is worthwhile refine the benchmarks to test for challenges specific to the end application.

In the process scheduling problem, an algorithm must be robust to random irregularities in load distribution. Reproducible noise is introduced in the large scale benchmark cases to model irregular loads and resource gradients that are not smooth and to assess their impact on load-balancing performance. Pseudo-random noise is added to the original benchmark field by applying a reproducible, discrete *noise mask* throughout the mesh. First, a two-dimensional grid of 100 x 100 random numbers uniformly distributed in the range [-10, 10] was created. Then, this grid was replicated throughout the mesh and the corresponding random value was added to each unit of the original benchmark field. Figure 32 displays the new gradient field that represents the noisy

resource distribution in all three large scale benchmark cases of the noise experiments. The original benchmark field is included for comparison. A detailed definition of the noise mask can be found in Appendix 3.



(a) (b)

Figure 32. (a) Original and (b) Noisy benchmark field.

The algorithms were evaluated on each set of benchmarks through multiple simulations. A swarm of processes—100 processes for standard benchmarks and 1000 processes for large scale benchmarks—is deployed in the environment. Processes are modeled as points, moving in the field as directed by the algorithm. A process "finds" a load trough when it is mapped within 10 CPUs of the center of the load trough, and it remains on that load trough for the remainder of the search. Simulations are terminated if all load troughs are utilized, if all processes are mapped to a trough, or after the maximum number of time steps is exceeded. The time steps of the simulations account for each process movement and each BRW process tumble. All processes move at a velocity of one unit per time step, and a BRW tumble is conducted in one time step. The simulated communication range of the GSO algorithm is 125 units.

The performance metrics of the benchmarks are the average number of load troughs found and the convergence times for each algorithm. The number of load troughs found at each time step is recorded for each simulation, and the average number of troughs found is calculated for all simulations. The 95% confidence intervals on the mean are calculated from the sample mean and standard deviation of all simulations to establish an error bar on the average performance. Two convergence times are defined to measure algorithm efficiency. The 75% convergence and 95% convergence metrics are the number of time steps to find an average of 75% and 95% of load troughs.

7.2 Sensitivity Analysis

Some algorithms, such as BRW, involve randomization as a primary ingredient of their approach. In order to compare the relative performance of algorithms, it is necessary to determine the minimum number of simulations that yield reliable metrics. A sensitivity experiment was conducted to determine this number. A BRW was simulated 5000 times on each benchmark of the standard and large scale sets. The mean and standard deviation of the number of load troughs found after 1000, 2000, 3000, 4000, and 5000 simulations were compared [34]-[35]. The standard deviation and average number load troughs found at each time step for each benchmark are shown in Figure 33. In all plots, the curves are too close to discern. Neither the average nor the standard deviation of load troughs found change significantly for more than 1000 simulations. Therefore, 1000 simulations were sufficient for performance comparisons.

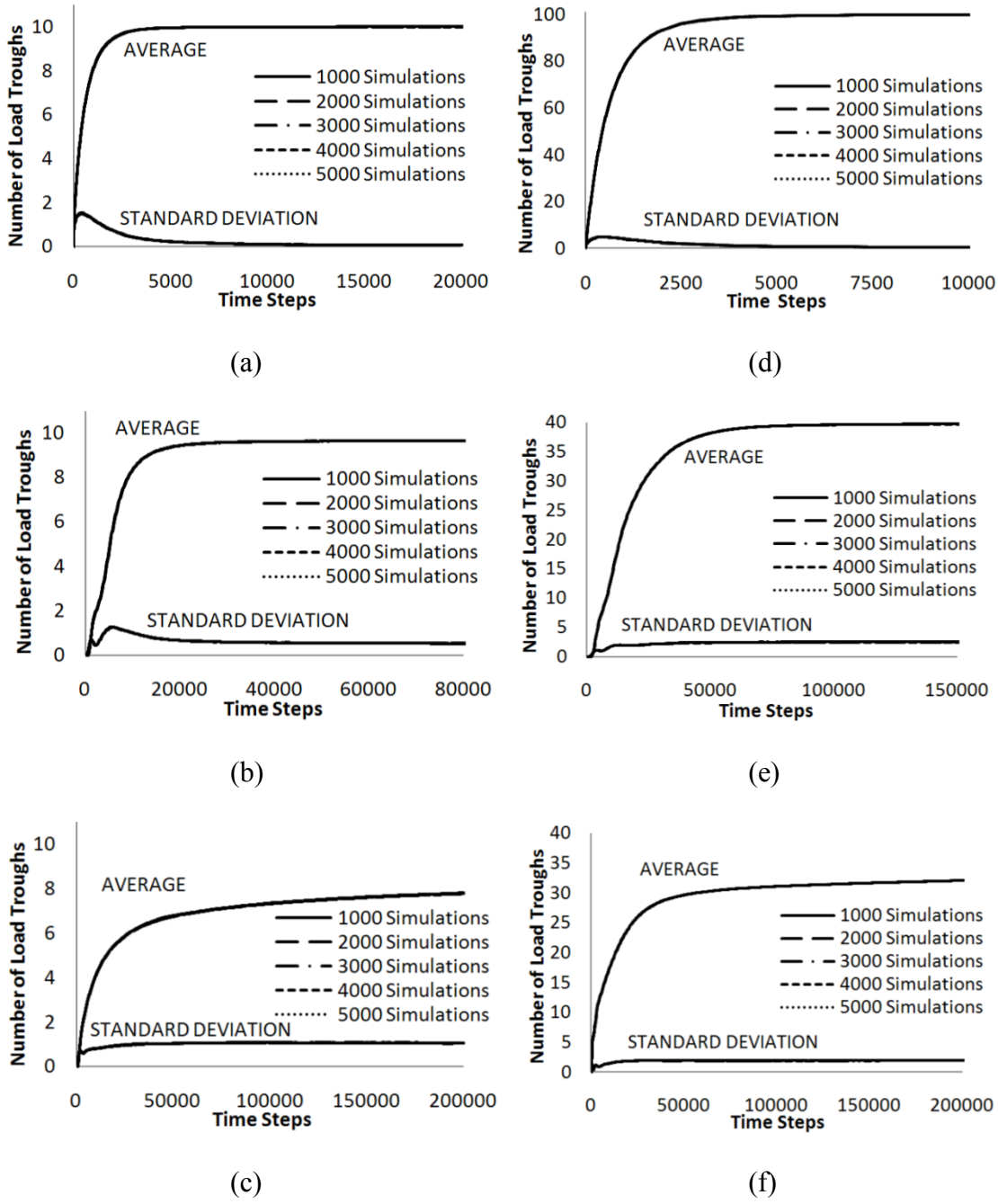
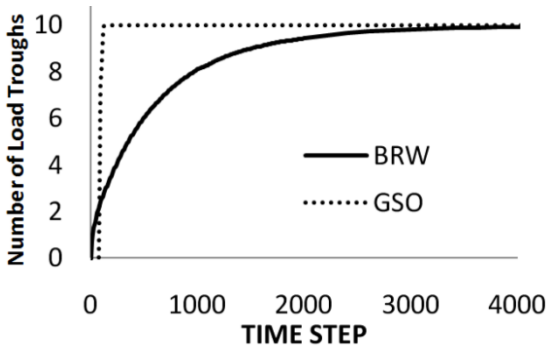


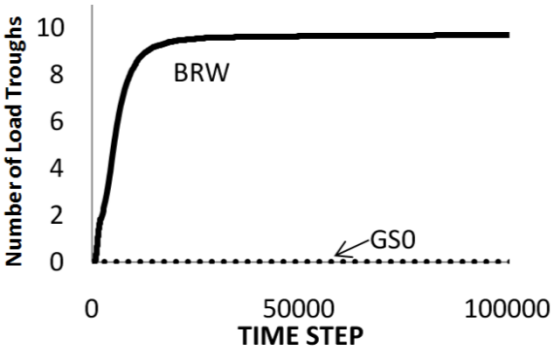
Figure 33. Average and standard deviation of the number of load troughs found v. time step for BRW on the (a) standard uniform, (b) standard point, (c) standard line, (d) large scale uniform, (e) large scale point, and (f) large scale line distributions.

7.3 Standard Benchmark Evaluation

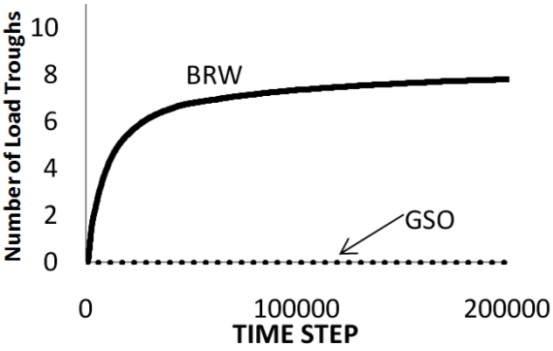
Figure 34 shows the average number of load troughs found at each time step for 1000 simulations of the BRW and GSO algorithm on each standard benchmark. In the uniform benchmark case, both algorithms locate all 10 load troughs on average. However, the GSO algorithm converges on the load troughs faster than BRW. GSO achieves 95% convergence in 110 time steps while the BRW achieves 95% convergence in 2200 time steps.



(a)



(b)



(c)

Figure 34. Average load troughs found v. time step for BRW and GSO on the standard (a) uniform, (b) point, and (c) line initial distribution benchmark cases.

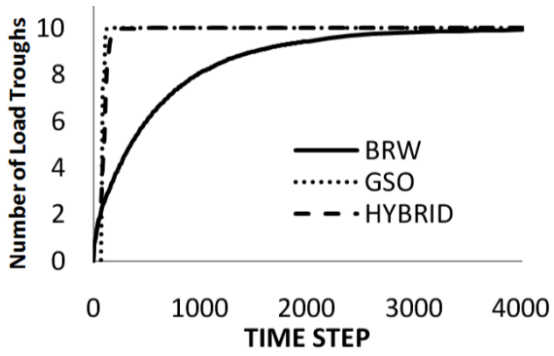
In the point and line benchmarks, the BRW locates an average of 9.7 and 8.5 load troughs in total, respectively. The algorithm does not locate all load troughs because a

number of processes exit the domain. Without prior knowledge of the domain, the algorithm provides no way to confine processes to the architecture. Thus, while the random nature of a BRW enables exploration for resources without initial coverage of the space, it does not guarantee localization of *all* load troughs.

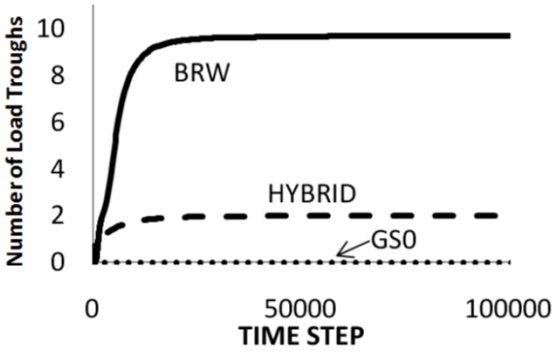
The GSO algorithm locates zero load troughs in both non-uniform cases. This failure is because GSO is handicapped by the dead space in the gradient field. Both the point and line distributions deploy the entire swarm in the dead space, so all processes have equal luciferin. The GSO algorithm requires that an process's neighbor have more luciferin than the agent itself. As a result, all processes are isolated and do not move.

To mollify the effect of dead space on the GSO performance, a hybrid GSO/BRW algorithm (HYBRID) was explored. The processes perform the same actions as the original GSO algorithm, unless a process's neighborhood is empty. In this case, it performs a single BRW step. Consistent with the parameters above, the HYBRID algorithm was tested for a communication range of 125 supplemented with a BRW with 10 unit step length and 10% bias.

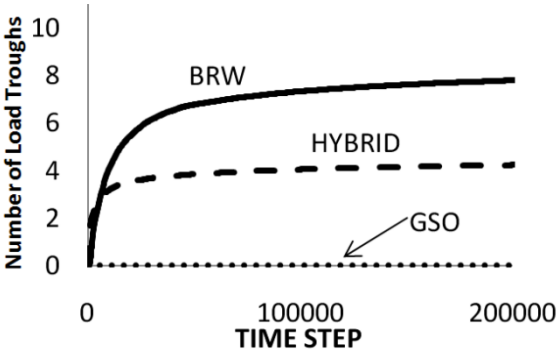
The average number of load troughs found at each time step for 1000 simulations of the HYBRID algorithm is plotted in Figure 35. The BRW and GSO algorithm curves are included in the figure for comparison. On the uniform benchmark, the HYBRID performance is very similar to the GSO algorithm. This similarity stems from the fact that the HYBRID only differs from GSO when an process's neighborhood is empty, which is seldom the case with uniform distribution.



(a)



(b)



(c)

Figure 35. Average load troughs found v. time step for BRW, GSO, and HYBRID on the standard (a) uniform, (b) point, and (c) line initial distribution benchmark cases.

The HYBRID algorithm fails to converge on 75% of the load troughs in either the point or line benchmarks. It appears the key to locating load troughs in the non-uniform

distributions is to explore the domain. The BRW step in the HYBRID algorithm enables exploration but only until the process locates a neighbor. Then, a strictly local search for resources is initiated through the GSO algorithm. As a result, the load troughs furthest from the initial mapping are never located, and the overall performance improvement is marginal.

Table 2 presents a summary of the performance metrics of the BRW, GSO, and HYBRID algorithms on the standard benchmark set. The table includes the number of time steps for 75% and 95% convergence and the total average number of load troughs found for each algorithm. The dashes indicate that the algorithm did not converge.

Table 2. Summary of Algorithm Performance on Standard Benchmarks

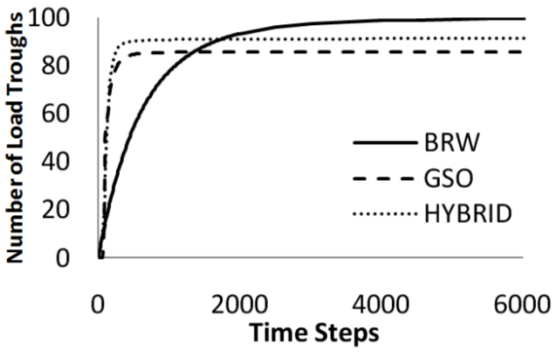
	Algorithm	**75% Convergence Time Steps**	**95% Convergence Time Steps**	**Average Load Troughs Found Total**
Uniform	BRW	820	2200	9.999 +/- 0.002
	GSO	100	110	10 +/- 0.0
	HYBRID	120	170	9.992 +/- 0.005
Point	BRW	8000	22600	9.716 +/- 0.032
	GSO	-	-	0.0 +/- 0.0
	HYBRID	-	-	2.000 +/- 0.041
Line	BRW	126400	-	8.549 +/- 0.060
	GSO	-	-	0.0 +/- 0.0
	HYBRID	-	-	4.637 +/- 0.043

These standard benchmarks represent the first stage of swarming algorithm analysis. Although the intention was to analyze the relative efficiency of the swarming

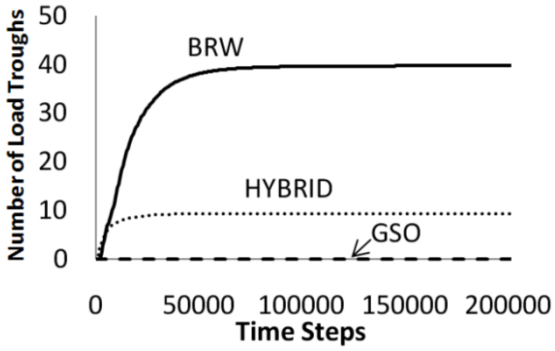
algorithms for process management, none of the algorithms were successful on all three benchmark cases. If the initial process distribution does not cover the domain, the algorithms do not locate all under-utilized clusters in the space. In addition, the dead space in the gradient field disabled the GSO algorithm. These surprising results emphasize the importance of including dead space and alternative initial distributions in testing algorithms for process management.

7.4 Large Scale Benchmark Evaluation

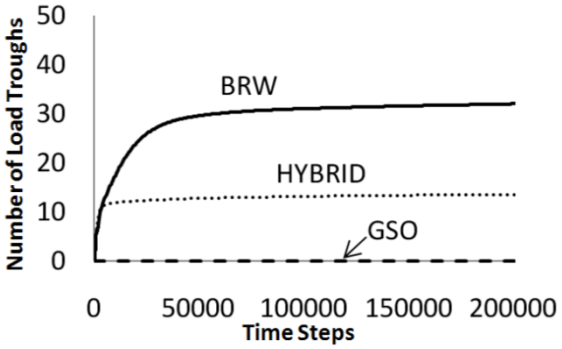
The next stage of analysis compares all three algorithms at large scale. Figure 36 displays the average number of load troughs found at each time step for 1000 simulations of the BRW, GSO, and HYBRID algorithms on the large scale benchmarks. In the uniform benchmark case, BRW locates an average of 99.95 load troughs in total. The GSO and HYBRID algorithms achieve 75% convergence faster than BRW, but only locate an average of 85.9 and 91.7 troughs in total, respectively.



(a)



(b)



(c)

Figure 36. Average load troughs found v. time step for BRW, GSO, and HYBRID on the large scale (a) uniform, (b) point, and (c) line initial distribution benchmark cases.

The degraded performance of the GSO and HYBRID algorithms on the uniform benchmark occurs because GSO favors load troughs with the most CPU resources

available. Consider a GSO or HYBRID process with two neighbors, each located near a distinct load trough. The process is more likely to move toward the neighbor near the larger trough and overlook the trough with fewer resources. As a result, the smaller trough may not be located, leaving CPU resources unutilized. The HYBRID algorithm locates more load troughs than GSO because many processes that were stalled with GSO can conduct an independent BRW to search for resources. In particular, the load troughs that border large regions of dead space are located more often by the HYBRID algorithm than the GSO algorithm by the processes mapped in the dead space.

None of the algorithms achieve 75% convergence on the point or line benchmark cases. The average number of load troughs found by BRW is 39.8 and 33.8, respectively. The algorithm performance suffers because processes exit the domain, like in the standard benchmarks. Also, multiple processes locate the same load trough. This effect is seen in all algorithms to an extent, but it is the most pronounced in BRW. The BRW is an independent algorithm in which a process stops at the first trough it locates. There is no way to guarantee that processes explicitly partition and locate all load troughs at large scale. The GSO algorithm locates zero load troughs in both the point and line benchmarks, and the HYBRID algorithm locates 9.3 and 14.1, respectively. These algorithms suffer for the same reasons explained in the standard benchmark analysis.

Table 3 presents a summary of the performance metrics of the BRW, GSO, and HYBRID algorithms on the large scale benchmarks. Perhaps the most significant finding in this study is that the all algorithms are less successful on the large scale benchmarks than they are on the standard benchmarks. Each algorithm located a larger percentage of the load troughs in the standard experiments than it did on the large scale experiments.

This fact holds for all three initial distributions and an equal ratio of the number of processes to load troughs. As the number of load troughs increases, it becomes more difficult to guarantee localization of all available resources.

Table 3. Summary of Algorithm Performance on the Large Scale Benchmarks

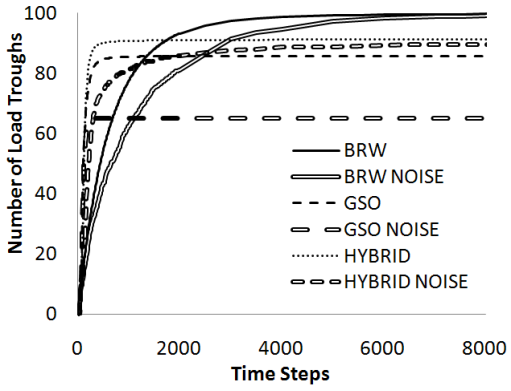
	Algorithm	75% Convergence Time Steps	95% Convergence Time Steps	Average Troughs Found Total
Uniform	BRW	930	2500	99.950 +/- 0.014
	GSO	210	-	85.922 +/- 0.110
	HYBRID	180	-	91.674 +/- 0.104
Point	BRW	-	-	39.758 +/- 0.158
	GSO	-	-	0.000 +/- 0.000
	HYBRID	-	-	9.344 +/- 0.092
Line	BRW	-	-	33.750 +/- 0.132
	GSO	-	-	0.0 +/- 0.0
	HYBRID	-	-	14.148 +/- 0.085

7.5 Noise Evaluation

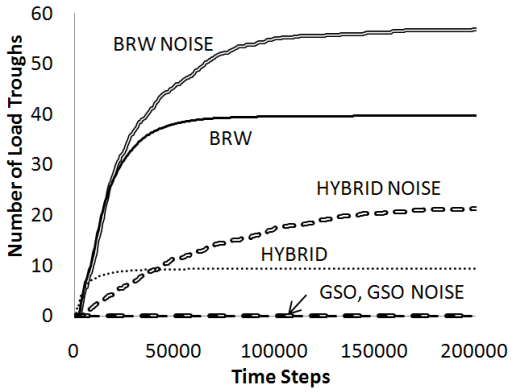
The last stage of comparative analysis evaluates the impact of noise on load-balancing performance. The performance of each algorithm is analyzed with respect to its performance in the *absence* of noise, rather than to the performance of competing algorithms. This perspective is intended to provide insights on the effect of noise on particular algorithm characteristics.

The average number of load troughs found at each time step for the noisy benchmarks is plotted in Figure 37. The curves for noiseless simulations are included for

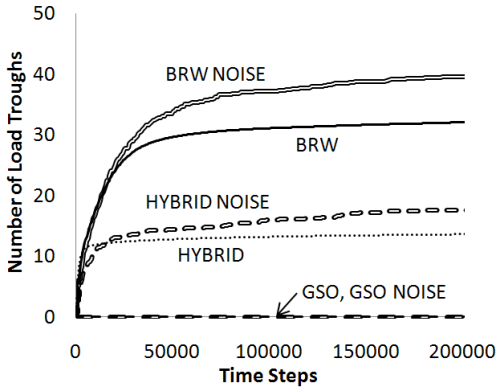
comparison. On the uniform benchmark, the BRW performance is comparable with and without noise. The algorithm is less efficient with noise, but ultimately locates all 100 load troughs on average. The HYBRID algorithm performance is also comparable. It is less efficient, achieving 75% convergence in approximately three times as many time steps but locates approximately the same number of load troughs in total on average. The GSO algorithm performance suffers, locating approximately 20 fewer load troughs on average. This result is attributed to the fact that some noise mimics local maxima in the benchmark field. A GSO process can be mapped to an isolated CPU with a large noise contribution and attract processes within range. In this way, subgroups of the swarm can be trapped on a cluster with few under-utilized processors. These subgroups are effectively removed from cooperative search and degrade the algorithm's overall load-balancing performance.



(a)



(b)



(c)

Figure 37. Average load troughs found v. time step for BRW, GSO, and HYBRID on the large scale (a) uniform, (b) point, and (c) line initial distribution benchmark cases *with noise*.

On the point and line benchmarks, the BRW algorithm, as well as the BRW component of the HYBRID algorithm, performs better in the presence of noise. It appears the noise emphasizes the random nature of BRW, which encourages processes to explore the architecture. Processes may be less susceptible to being drawn to the first load trough they encounter. The GSO algorithm locates zero load troughs in the noisy point and line benchmarks. Again, the GSO process attraction to isolated CPUs with large noise contributions render it unable to navigate beyond the dead space. A summary of the large scale performance on the original and noisy benchmarks is presented in Table 4.

Table 4. Comparison of Algorithm Performance on the Original and Noisy Large Scale Benchmarks

	Algorithm	**75% Convergence Time Steps**		**95% Convergence Time Steps**		**Average Load Troughs Found Total**	
		Original	Noise	Original	Noise	Original	Noise
Uniform	BRW	930	1500	2500	4500	100	100
	GSO	210	-	-	-	85.9	65.0
	HYBRID	180	530	-	-	91.7	91.1
Point	BRW	-	-	-	-	39.8	57.8
	GSO	-	-	-	-	0	0
	HYBRID	-	-	-	-	9.3	22.8
Line	BRW	-	-	-	-	33.8	42.7
	GSO	-	-	-	-	0	0
	HYBRID	-	-	-	-	14.1	18.5

7.6 Summary of Process Management Requirements

After several stages of analysis, none of these algorithms were successful in locating all load troughs in the benchmarks [34]-[35]. However, several measures have been identified to improve performance of a tailored process scheduling algorithm:

- The algorithm must include a mechanism to search for resources when there is no load gradient to exploit. This feature averts the GSO handicap in the non-uniform benchmarks.

- Processes should balance local search for resources with broad exploration of the architecture. The benchmark simulations highlight the tradeoffs between regimes of local search and broad exploration. With uniform initial distribution, GSO emphasizes local search and locates load troughs more efficiently than BRW. In contrast, the BRW algorithm is more exploratory and locates more under-utilized resources with non-uniform initial distribution. A viable approach must incorporate exploration of the *entire* search space.

- Processes should be confined within the dimensions of the architecture, taking appropriate actions at the boundaries. This measure prevents processes from exiting the search domain, which was damaging to the BRW algorithm performance. However, this requires prior knowledge of the architecture domain.

- Processes should continue to aid other processes of the swarm after a load trough has been located. This will prevent multiple processes from converging on the same load trough, a significant problem on the large scale benchmarks.

Chapter 8 DIFFUSE Algorithm for Scheduling rMP Processes[6]

This chapter presents a novel algorithm, DIFFUSE, for scheduling resilient processes on distributed architecture. It is inspired by the notions of heat diffusion and robotic swarming and may form the core of operating system support for a resilient process scheduler. Heat diffusion is emulated to distribute processes across the computer architecture [75]-[76]. Robotic swarming techniques are used to manage resilient processes. DIFFUSE incorporates lessons learned from benchmark analyses as well as concepts from the survey of swarming algorithms to improve on load-balancing performance of the original benchmarks.

8.1 Design and Implementation

Figure 38 presents the DIFFUSE algorithm, which includes two distinct modes of swarm control: SEARCH mode and DISPERSE mode. In SEARCH mode, processes conduct an independent *local* search for resources using the BRW algorithm with a 10% bias [88]. In DISPERSE mode, the processes react to virtual repulsive forces from neighbors and spread across the architecture [97], [100]. Each process begins in DISPERSE mode to promote initial coverage of the architecture (1). Each iteration, the process checks for a load trough in its location (2) and shares its position and mode with all neighbors within communication range R (3). The process updates its mode based on

[6] Significant portions of this chapter have been published or submitted for publication in the following:
- K. McGill and S. Taylor. "DIFFUSE algorithm for robotic multi-source localization", *Proc. of IEEE Int. Conf. on Technologies for Practical Robot Applications (TePRA)*, April 2011.
- K. McGill and S. Taylor, "Operating System Support for Resilience," *IEEE Transactions on Reliability*, submitted for publication.

the most recent communications (4) and implements either the SEARCH (5) or DISPERSE (6) algorithm. As a result, the processes' local interactions and random motions emerge as a collective *diffusion* of the swarm across the architecture.

```
mode = DISPERSE;                          /*1*/
while(troughs_not_found)  {
    check_for_troughs();                  /*2*/
    neighbor_communication();             /*3*/
    update_mode();                        /*4*/
    if(mode = = SEARCH)  {                /*5*/
        BRW();
    }
    if(mode = = DISPERSE)  {              /*6*/
        calculate_net_force(net_force);
        move(net_force);
    }
}
```

Figure 38. DIFFUSE algorithm pseudo-ocode.

Specific conditions prompt a process to change its mode [36]. If the process finds an *unoccupied* load trough, it will remain there and signal DISPERSE mode to its neighbors. If the process is in DISPERSE mode and in the same position as the last iteration, it changes to SEARCH mode. If the process is in SEARCH mode and communicates with a neighbor in DISPERSE mode, the process changes to DISPERSE mode. Finally, if the process leaves the architecture domain at any time, it enters DISPERSE mode and executes a protocol to return to the domain.

These conditions are designed to combine the SEARCH and DISPERSE modes for effective load-balancing. The DISPERSE mode signal originates from a process that first locates a load trough, and the signal propagates throughout the swarm. The switch

to SEARCH mode occurs when a process experiences no net force, either because it is equally spaced between neighbors or because there are no other processes within range. In the first case, the SEARCH initiates local search of the process's region. In the second case, the SEARCH enables independent exploration. Upon switching to SEARCH mode, the process stays in SEARCH mode for a minimum number of BRW steps to permit sufficient local search of the area.

The DISPERSE mode algorithm incorporates a tactic from the PSO swarming algorithm [97]. It creates a virtual electrostatic potential field in the architecture [100]. Unlike the BRW, GSO, and HYBRID algorithms, the DIFFUSE algorithm includes *a priori* knowledge of the architecture boundaries in order to confine processes. The processes and the boundary of the system are modeled as positively charged particles that repulse each other. The net force felt by an individual process, $\mathbf{F}_i$, is a summation of the individual repulsive forces from all neighbors, *j*, within communication range.

$$\mathbf{F}_i = -\sum_j \frac{k}{r_{ij}^2}\mathbf{n}_{ij} \tag{6}$$

In (6), r_{ij} is the Euclidian distance between processes *i* and *j* in the architecture mapping, k is a constant, and $\mathbf{n}_{ij}$ is a unit vector pointing from process *i* to process *j* [100]. If the process is within range of the boundary of the architecture, the closest point on the boundary is treated as another single repulsive force on the process. The length of the process's step is equal to the magnitude of the net force up to a maximum step size.

8.2 Analysis and Discussion

The DIFFUSE algorithm exhibits several characteristics prevalent in related work in heat diffusion and robotic swarming algorithms. It shares attractive properties with

heat diffusion in that it uses a local iterative scheme, requires no global communication, and provides global convergence for complex large-scale load distributions. However, to achieve resilience, two additional process scheduling requirements are included. Process replicas must maintain locality and be mapped to different processors.

The resiliency requirements are achieved through the emergent properties of robotic swarming techniques [95], [105]-[108]. In the DIFFUSE algorithm, the potential field in the search domain corresponds to a map of processes in the architecture. Repulsive forces are inversely proportional to the distance between processes and are limited by communication. These qualities provide the emergent behaviors of the algorithm. Processes diffuse as a result of the virtual potential field dispersing processes uniformly throughout the system. However, processes maintain locality because repulsion weakens with distance, and the process step size is limited. The swarm dynamics prevent processes from repelling too far and violating locality. Finally, strong repulsion at close range prevents processes from being mapped to the same processor.

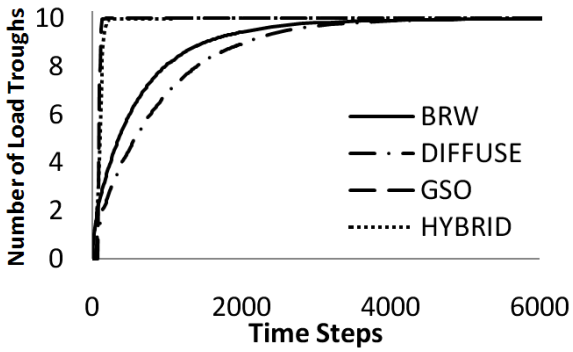
8.3 Experimental Evaluation

To compare the performance of the DIFFUSE algorithm to other candidates, it is simulated on all three benchmark sets presented in Chapter 7. The simulation parameters are consistent with the original benchmarks. Unless otherwise specified, the DIFFUSE algorithm maintains a communication range of $R = 200$ units.

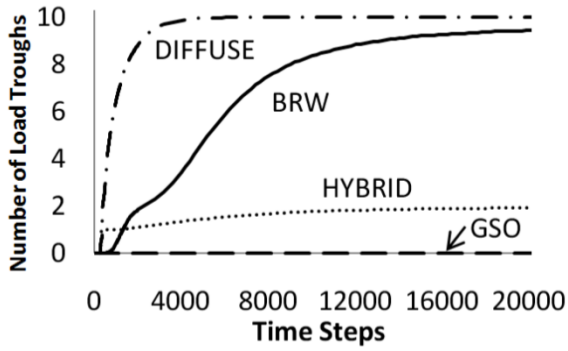
8.3.1 Standard Benchmarks

The average number of load troughs found at each time step for 1000 simulations of the DIFFUSE algorithm on each standard benchmark case is plotted in Figure 39. The BRW, GSO, and HYBRID algorithm curves are included in the figure for comparison.

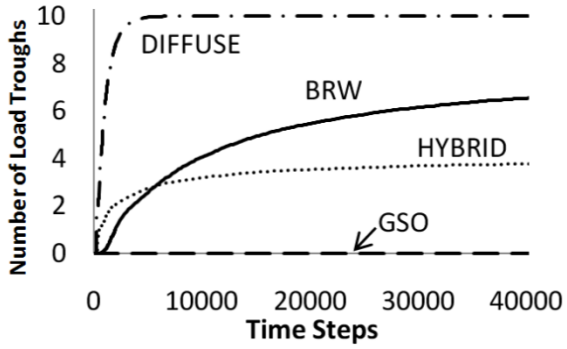
Like BRW and GSO, DIFFUSE locates all ten load troughs on the uniform benchmark case. The efficiency of the algorithm is comparable to BRW. Because the uniform initial distribution provides coverage of the architecture from the beginning, there is little advantage to the DISPERSE mode of DIFFUSE. The GSO and HYBRID algorithms achieve 95% convergence faster than the DIFFUSE algorithm. This result occurs because the GSO and HYBRID algorithms are greedier, emphasizing local search. The processes executing these algorithms always move toward greater available resources. In contrast, the DISPERSE mode of the algorithm can slow the localization of load troughs when the swarm is already diffused by preventing processes from executing local search.



(a)



(b)



(c)

Figure 39. Average load troughs found v. time step for DIFFUSE, BRW, GSO, and HYBRID on the standard (a) uniform, (b) point, and (c) line initial distribution benchmark cases.

The DIFFUSE algorithm also locates all ten load troughs on the point and line benchmark cases, unlike the other candidates. In addition, it achieves 75% and 95% convergence faster than BRW on both benchmarks. Because the initial process distribution is non-uniform, the DISPERSE mode accelerates localization of load troughs by enforcing coverage of the domain from the start. Subsequently, under-utilized resources that are distant from the initial process mapping are located quickly.

In addition to the standard benchmark analysis, the effect of communication range on DIFFUSE algorithm performance is investigated by including six ranges of 10, 50, 100, 150, 200, and 250 units. Figure 40 displays the effect of communication range on the algorithm performance. The figure shows 95% convergence times as a function of range on the uniform, point, and line benchmarks.

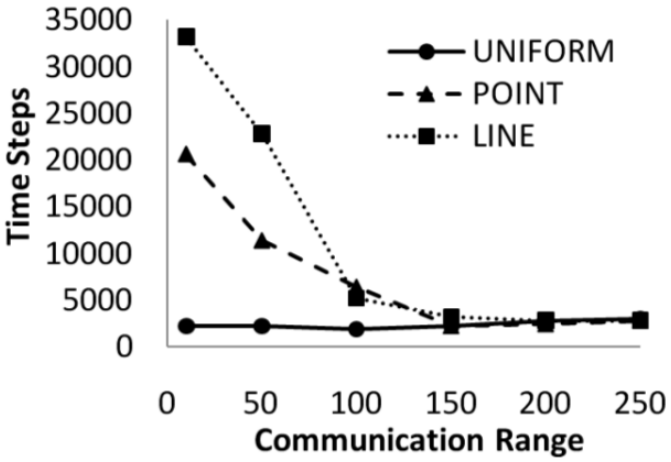
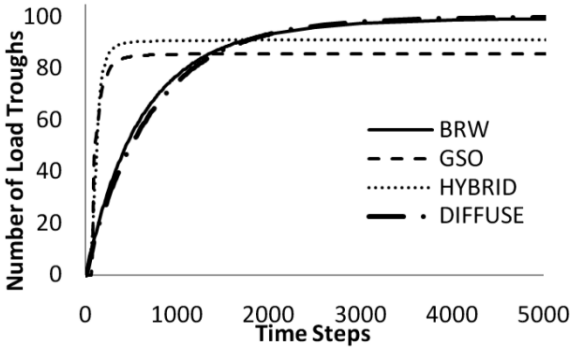


Figure 40. Time steps to 95% convergence v. communication range for the DIFFUSE algorithm on uniform, point, and line initial distributions.

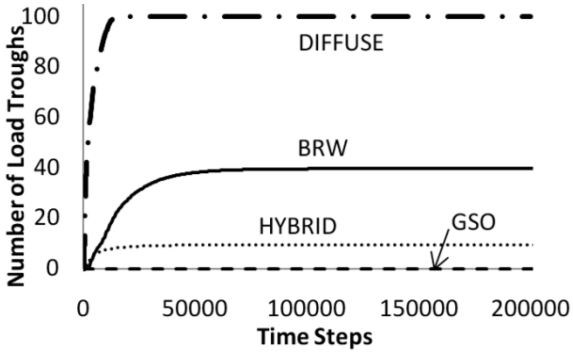
The DIFFUSE performance is roughly independent of communication range with the uniform initial distribution. This reinforces the fact that DIFFUSE provides no advantage when the initial distribution covers of the domain. In the non-uniform benchmarks the DIFFUSE algorithm performance improves as the communication range increases, up to a critical range of approximately 200 units. This distance is the minimum range that enforces diffusion across the entire domain. Above this critical range, DIFFUSE performance is comparable on all three benchmarks. Effectively, at the critical range, the algorithm nullifies the negative impact of non-uniform distributions on load-balancing performance.

8.3.2 Large Scale Benchmarks

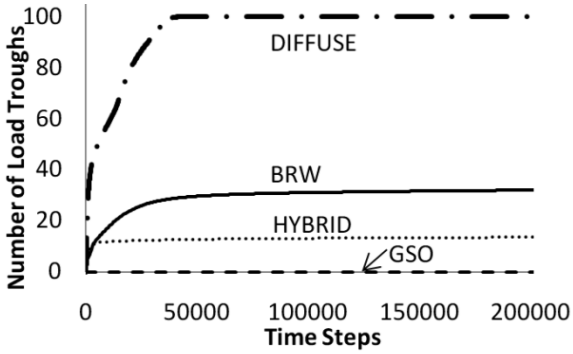
Figure 41 shows the average number of load troughs found at each time step for the DIFFUSE and other candidate algorithms on the large scale benchmarks. On the uniform benchmark, the DIFFUSE algorithm is the only algorithm to locate all 100 load troughs in every simulation. The efficiency is comparable to the BRW algorithm but less than GSO and HYBRID. These results are consistent with the standard benchmarks.



(a)



(b)



(c)

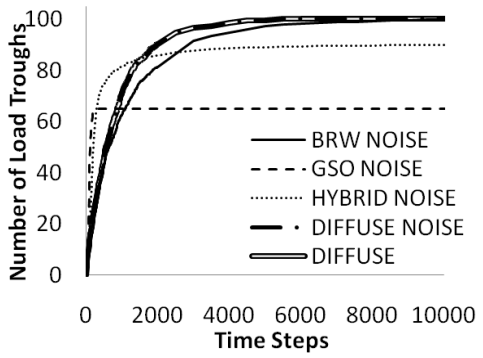
Figure 41. Average load troughs found v. time step for DIFFUSE, BRW, GSO, and HYBRID on the large scale (a) uniform, (b) point, and (c) line initial distribution benchmark cases.

DIFFUSE locates all 100 load troughs on the point and line benchmark cases, in

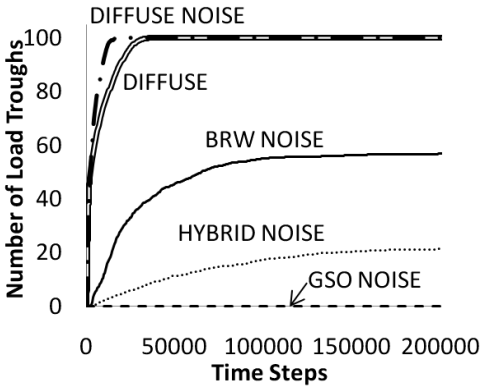
contrast to the other algorithms. Recall the reasons for the BRW algorithm failure on the non-uniform benchmarks: processes exiting the domain and multiple processes locating the same load trough. The DIFFUSE algorithm uses the potential field to prevent both of these problems. Processes are repulsed from the architecture boundary which prevents them from exiting the domain. Clearly, the DIFFUSE algorithm's knowledge of the domain provides an advantage. Similarly, process repulsion prevents multiple processes from converging on the same trough. These features enable the DIFFUSE algorithm to locate *all* load troughs at large scale.

8.3.3 Noise Benchmarks

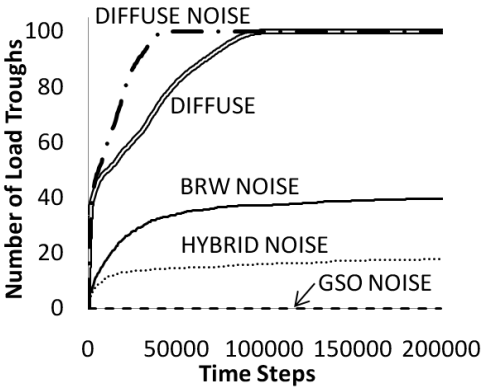
The final evaluation of the DIFFUSE algorithm assesses the impact of noisy resource gradients on search performance. Ten searches of DIFFUSE on the noisy uniform, point, and line benchmark cases are simulated. The average number of load troughs found at each time step for each benchmark case is plotted in Figure 42. The BRW, GSO, and HYBRID noise curves and the DIFFUSE curve from the simulations without noise are included in the figure for comparison. The DIFFUSE algorithm exhibits the best overall performance of all four algorithms. Like the experiments without noise, it is the only algorithm that locates all 100 load troughs in all three benchmarks.



(a)



(b)



(c)

Figure 42. Average load troughs found v. time step for DIFFUSE, BRW, GSO, and HYBRID on the large scale (a) uniform, (b) point, and (c) line initial distribution benchmark cases with noise.

The DIFFUSE algorithm performance with noise is comparable to its performance without noise in the uniform benchmark. In the non-uniform benchmarks, DIFFUSE is more efficient with noise. The dispersion of processes is dominated by swarm dynamics and independent of the load distribution. Consequently, the swarm as a whole is not susceptible to local maxima or isolated irregularities in load. The noise does not hinder the diffusion of processes, which accelerates load trough localization for DIFFUSE. The efficiency gains of the algorithm in the non-uniform benchmarks are consistent with prior studies. DIFFUSE uses the BRW algorithm in its local search mode and reflects the enhanced performance of the BRW algorithm in the presence of noise.

8.4 Summary and Conclusions

The DIFFUSE algorithm shows potential for the process scheduling problem. Unlike the other candidates explored, DIFFUSE locates all load troughs on all benchmark simulations. In particular, it outperforms the other algorithms when the initial process distribution is non-uniform by diffusing processes across the architecture, using only limited range communication.

DIFFUSE also proves to be robust to noise; the dispersion of processes is dominated by swarm dynamics rather than the local load statistics. The local search for resources is conducted by processes individually. This combination emulates diffusion on large scales and load-balancing on smaller scales.

The algorithm's emergent behavior provides locality properties to supplement load-balancing for resilient process management. However, there is room for further development to optimize the resilient qualities. The tradeoffs between maintaining process *locality* for reduced message delays and process *separation* for resilience warrant

exploration. A better understanding of these tradeoffs in process interactions is particularly valuable in the optimization of DIFFUSE for the management of multiple concurrent applications.

Chapter 9 Reliability Analysis of rMP Applications

Resilience is defined as the ability to provide and maintain an acceptable level of service in spite of a range of faults and attacks [31]. This thesis presents a variety of mechanisms and policies to achieve application resilience to process failures. The remaining task is to assess the reliability of the rMP technology. By constructing a set of analytical tools, it is possible to predict reliability to a variety of faults and attacks and weigh the merits of the approach for target applications.

9.1 Definitions and Metrics

There are a number of terms and metrics in the fault tolerant and computer security communities with unclear definitions and usage. Several fundamental definitions are provided to clarify this discussion.

A **fault** is a flaw in a system whose presence may or may not lead to a failure [40]. In order to cause a failure, the fault condition must be executed by a system component. An **error** is a deviation between the expected behavior and the actual behavior of a component *internal* to the system [40]. Errors occur during execution, due to the activation of a fault. A **failure** is an event in which an observable system behavior deviates from its specification [40]. Failures are events that are detectable at the system boundary. For example, process failures are detected through the rMP mechanisms. Generally, there is a progression in which a fault is activated, causing an error, which may result in a failure.

A different set of terms is used in the computer security community. A **vulnerability** is defined as a flaw or weakness in system security procedures, design,

implementation, or internal controls [109]. A vulnerability can be unintentionally triggered, in the case of an **accident**, or intentionally exploited, in the case of an **attack**. A **threat** is defined as an accident or attack that triggers a vulnerability, which may cause a security breach [109]. A **security breach** is a violation of the system security specification. Like faults, vulnerabilities do not always propagate into security breaches, but this analysis assumes that they do.

There are a number of metrics used to describe the desirable attributes of mission-critical systems. The minimal set of attributes is a question of ongoing debate. However the working definitions of terms that pertain to this thesis are provided by Trivedi *et al.* [110]. In addition, each term is a discussed in the context of application resilience.

Reliability is the probability that a system performs a specified service throughout a specified interval of time [111]. This is a measure of the continuity of service, or the probability that the system has not failed once since it started. Generally, individual component failures are allowed during this interval, but not whole system failures. Reliability serves as the primary metric for this analysis. However, it is often more natural to refer to the probability of failure of a system over the specified interval. The reliability, *R(t)*, and probability of failure, *F(t)*, always sum to unity, as defined in (7).

$$R(t) + F(t) = 1 \tag{7}$$

Availability is the ability of the system to perform its prescribed function at a specific instant of time or over a specified period of time. The metric is usually expressed as a ratio of the units of time when service was available to the specified service period. Availability is closely related to reliability. However, it depends on the

service provided by the application. For instance, in a manager-worker application, the failure and regeneration of a worker process might not compromise availability. However, if a manager process fails, the application service might become temporarily unavailable. Because of these discrepancies, availability is not considered in this analysis.

Survivability is the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents [112]. The mathematical definition of survivability is a separate research question [113], [114], [115]. According to the working group on Network Survivability Performance, survivability depicts the time-varying system behavior after a failure, attack, or accident occurs [116]. A thorough treatment of the survivability metric is beyond the scope of this thesis. It emphasizes analysis on time scales of a failure event. Instead, reliability is used to describe the survivability on the scale of the service interval.

Finally, a few parameters used to define the fault models are presented for reference. The **Mean Time Between Failure (MTBF)** is an estimate of the elapsed time between failures in a component or system. A straightforward interpretation is the operational time divided by the number of failures. The **failure rate (λ)** is the inverse of the MTBF (1/MTBF), expressed in failures per unit time.

9.2 rMP Application Model

This analysis aims to characterize the reliability of the *approach* presented in this thesis. An ideal implementation of the prototype is assumed in which the operating system mechanisms provide application resilience as designed. All failures are detected through communication timeouts and message comparison, processes are reliably

regenerated, and the process management strategies ensure that process replicas are mapped to different hosts. Thus, reliability analysis is conducted with respect to the application; the reliability of the rMP technology and the operating system is assumed.

The following application model is used. An application consists of p processes with a level of resiliency r, so the resilient implementation includes a total of $p*r$ processes. Each process is mapped to a different processor. Assume that the application will fail if all r replicas of a particular process group experience complete process failures at the same time *or* if greater than $(r\text{-}1)/2$ replicas of a particular process group experience Byzantine process failures at the same time.

Although application failure requires the replicas be failed at the same time, it is not necessary that the *root causes* of process failures are simultaneous. In the rMP technology, process failures are detected in *msgrecv()* calls. As a result, there is a failure time interval between communications during which replica failures can be considered simultaneous. This interval begins after a consistent *msgsend()* call and continues through subsequent computation, the next *msgsend()* call, failure detection, and process regeneration. Portions of this interval can be estimated from the performance benchmarks, but the time between subsequent *msgsend()* calls is strictly application dependent. For this analysis, it is assumed that processes communicate relatively frequently, and the failure time interval is 30 seconds. This 30 second interval is a conservative estimate. It *overestimates* the probability of failure for the distributed exemplars in which communication occurs more frequently.

To compare the reliability of rMP applications to the fault tolerant community, reliability analysis is also included for applications using static replication without

process recovery. It is assumed that an application using static replication will fail with the same number of replica failures as rMP applications, but now those failures can occur over the duration of the computation. For this analysis, the failure time interval is the required service interval of the application; an interval of 20 minutes is selected as an initial pass. Otherwise, the same parameters are used for static replication as rMP applications.

The reliability is considered as a function of the level of resiliency for three application sizes: 100, 1000, and 10,000 processes. This number of processes may correspond to a single application or multiple applications running simultaneously. One process is mapped to each processor of the system, so the number of processors used increases with the level of resiliency. In order to conduct analysis in the context of mission-critical applications, the *maximum probability of failure acceptable for mission-critical applications is defined as* 10^{-6} [117].

9.3 Analytical Model of Reliability

An analytical model is constructed to determine the probability of application failure for rMP applications. Given the probability of a single process failure, P_f, over interval t, the probability of failure of m processes in the same interval, F_m , is defined by (8).

$$F_m = P_f^m \tag{8}$$

The number of process failures required for the failure of an entire process group depends on the process failure mode. For simplicity, it is assumed that applications experience either complete or Byzantine process failures, but not both. Using this assumption, the

number of replica failures required for process group failure, m, is defined by (9) for complete failures, m_C, and by (10) for Byzantine failures, m_B.

$$m_C = r \tag{9}$$

$$m_B = floor(\frac{r-1}{2} + 1) \tag{10}$$

Thus, (8) provides the probability of failure of a single process group for the appropriate definition of m. Conversely, the reliability of a single process group, R_m is defined by (11),

$$R_m = 1 - F_m = 1 - P_f^m \tag{11}$$

However, for an application to be reliable, all process groups must be reliable. The probability that all p process groups are reliable, R_{app}, is defined by (12).

$$R_{app} = (1 - P_f^m)^p \tag{12}$$

Finally, the probability of application failure, F_{app}, is defined by (13).

$$F_{app} = 1 - R_{app} = 1 - (1 - P_f^m)^p \tag{13}$$

By substituting the appropriate definitions of m, the probability of application failure due to complete process failures, $F_{app,C}$, and the probability of application failure due to Byzantine process failures, $F_{app,B}$, are defined by (14) and (15), respectively.

$$F_{app,C} = 1 - (1 - P_f^r)^p \tag{14}$$

$$F_{app,B} = 1 - (1 - P_f^{(floor((r-1)/2+1))})^p \tag{15}$$

In order to determine the application failure probabilities defined in (14) and (15), the probability of failure for a single process is provided by the fault or threat model under consideration.

9.4 Fault and Threat Models

A number of fault and threat models are presented to provide the basis for quantitative analysis of reliability. These models are not meant to be exhaustive of the threat space. The space is too large for a thorough treatment to be practical. Rather, they lend a starting point to examine the strengths and weaknesses of the approach. Each model is used to determine the probability of failure for a single process in the environment to input in the analytical model of application reliability.

For all models, an architecture of 10,000 distributed hosts is assumed. Each host contains at least 8 processors, so the system has the capacity to run up to 8 replicas of 10,000 processes simultaneously. Each host includes application software which may contain vulnerabilities. However, it is assumed that any software vulnerabilities are restricted to the *application* layer of the host. The operating system software is reliable.

9.4.1 Independent and Identically Distributed Fault Model

The most common fault model applied in the fault tolerant community assumes independent and identically distributed faults in hardware components [25], [118]. This model offers a simple reference case for analysis. Figure 43 provides a summary of the key assumptions, parameters, and equations of the independent fault model analysis.

Independently and Identically Distributed Faults

Goal: Determine the probability of application failure on a scalable multi-processor architecture with independently and identically distributed faults.

Assumptions:
Independent and identically distributed faults
Exponential processor failure distribution
Faults are transient or repaired manually.
A fault causes complete failure of the process that activates it when it occurs.

Parameters:
Processor Mean Time Between Failure (MTBF): 5 years
Processor Failure rate (λ): 0.2 failures/year
Failure time interval (t): 30 seconds for rMP applications
20 minutes for static replication

Equations:
Probability of application failure from complete process failures:

$$F_{app,C} = 1 - (1 - P_f^r)^p \quad (14)$$

Probability of processor failure with exponential failure distribution:

$$P_f = 1 - e^{-\lambda t} \quad (16)$$

Figure 43. Summary of assumptions, parameters, and equations for the analysis of the probability of application failure in the presence of independent and identically distributed faults.

The model considers faults caused by processor failures, in which all processors have an equal failure rate. An exponential failure distribution is assumed for the processors, such that the probability of a processor failure, P_f, over a time interval t is defined by (16).

$$P_f = 1 - e^{-\lambda t} \tag{16}$$

Unfortunately, there is no clear consensus of general processor MTBF in the literature, but a conservative estimate of 5 years is commonly used [118], [119], [120], [121]. The time interval, *t*, corresponds to the failure time interval for rMP and static replication applications, 30 seconds and 20 minutes, respectively. This model assumes that all faults are transient or may be repaired manually. However, the outcome of a fault is complete process failure for the process that activates it when it occurs.

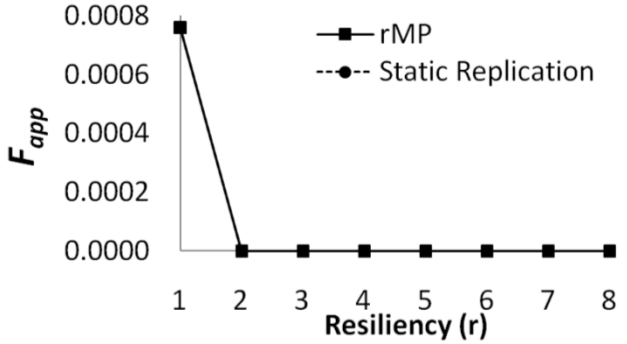
For scalable multi-processor architectures, it is often worthwhile to consider the system MTBF. For a system with *n* processors and independent and identically distributed faults, the system MTBF, $MTBF_{SYS}$, is defined by (17).

$$MTBF_{SYS} = \frac{MTBF}{n} \tag{17}$$

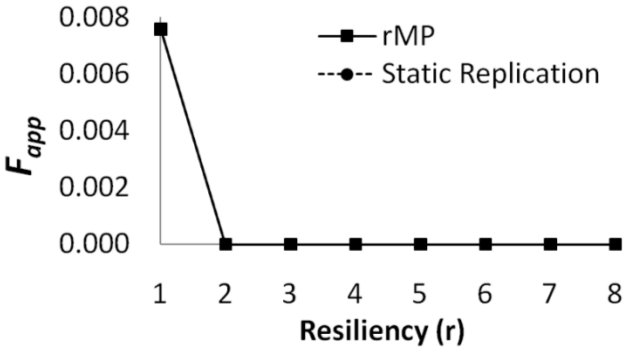
For systems with large numbers of processors, failures become more frequent. For example, the system MTBF for a platform with 100,000 processors, each with an individual 5 year MTBF, is approximately 26 minutes. At this frequency, failures are guaranteed to interrupt long running applications and represent an intolerable threat to any mission-critical application. This phenomenon is a major driving force for fault tolerance to independent and identically distributed faults as systems increase in scale.

Figure 44 shows the probability of application failure due to complete process failures caused by independent faults for rMP and static replication. Not surprisingly, the probability of failure is very low for both replication strategies. For all applications, the probability of application failure is nearly indiscernible, below 10^{-6} with a single replication ($r = 2$). These results are consistent with the expectation that a properly functioning system would not be likely to cause application failure. However, note that a

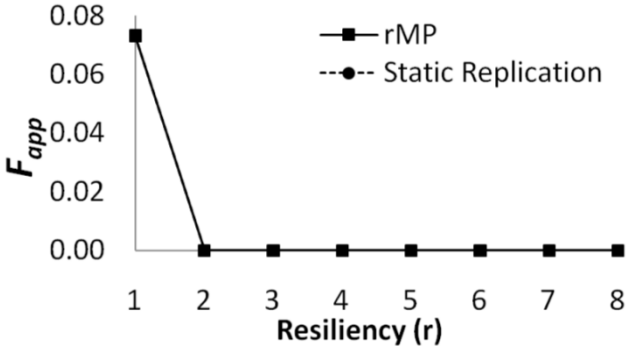
system with 10,000 processes and no fault tolerant strategy has a 7.3% chance of failure, which exceeds what is acceptable for mission-critical applications.



(a)



(b)



(c)

Figure 44. Probability of application failure due to complete process failures caused by independent faults for rMP and static replication with increasing levels of resiliency for applications with (a) 100, (b) 1000 , and (c) 10,000 processes.

9.4.2 Correlated Fault Model

In reality, faults are not usually independent and identically distributed. Hardware faults are often correlated, due to environmental conditions, temperature, manufacturing defects, incorrect configuration, etc. Other correlated failures are caused by software, storage systems, networks, human error, and a number of unknowns. In addition, there is potential for coordinated attacks to cause correlated faults in critical computer systems. The analysis of correlated faults on a variety of computing platforms is an active area of research [122]-[124]. These studies use existing failure logs to model the observed failure patterns of multi-processor systems. One pertinent finding is that failures tend to be periodic and strongly correlated in time. Often there are failure peaks or bursts during which multiple failures occur in relatively short time intervals [124]. To assess the reliability of the rMP technology in these real-life correlated failure scenarios, a model of correlated faults is developed. Figure 45 provides a summary of the key assumptions, parameters, and equations of the correlated fault analysis.

Correlated Faults

Goal: Determine the probability of application failure on a scalable multi-processor architecture with correlated faults.

Assumptions:
Bursts of multiple hardware faults occur in short intervals *during which* there are:

- Independently distributed faults
- Exponential processor failure distributions

Faults are transient or repaired manually.
A fault causes complete failure of the process that activates it when it occurs.

Parameters:
Processor Mean Times Between Failure (MTBF): 10.6 hours and 4.7 hours
Processor Failure rates (λ): 826 failures/year and 1883 failures/year
Failure time interval (t): 30 seconds for rMP applications
20 minutes for static replication

Equations:
Probability of application failure from complete process failures:

$$F_{app,C} = 1-(1-P_f^r)^p \quad (14)$$

Probability of processor failure with exponential failure distribution:

$$P_f = 1-e^{-\lambda t} \quad (16)$$

Figure 45. Summary of assumptions, parameters, and equations for the analysis of the probability of application failure in the presence of correlated faults.

The data and failure analysis obtained from two platforms in the literature is leveraged for this analysis [123]-[124]. These platforms were selected because they represent distributed systems most likely to utilize the rMP technology: high performance computing (HPC) and grid computing systems. The HPC data is obtained from 22 HPC systems at the Los Alamos National Laboratory (LANL). The grid representative is Grid 5000, a scientific instrument of geographically distributed systems. Table 5 summarizes

the key platform characteristics, including the type of distributed system, the number of processors, and years during which the data was collected [123]-[124].

Table 5. Platform Data Set Characteristics

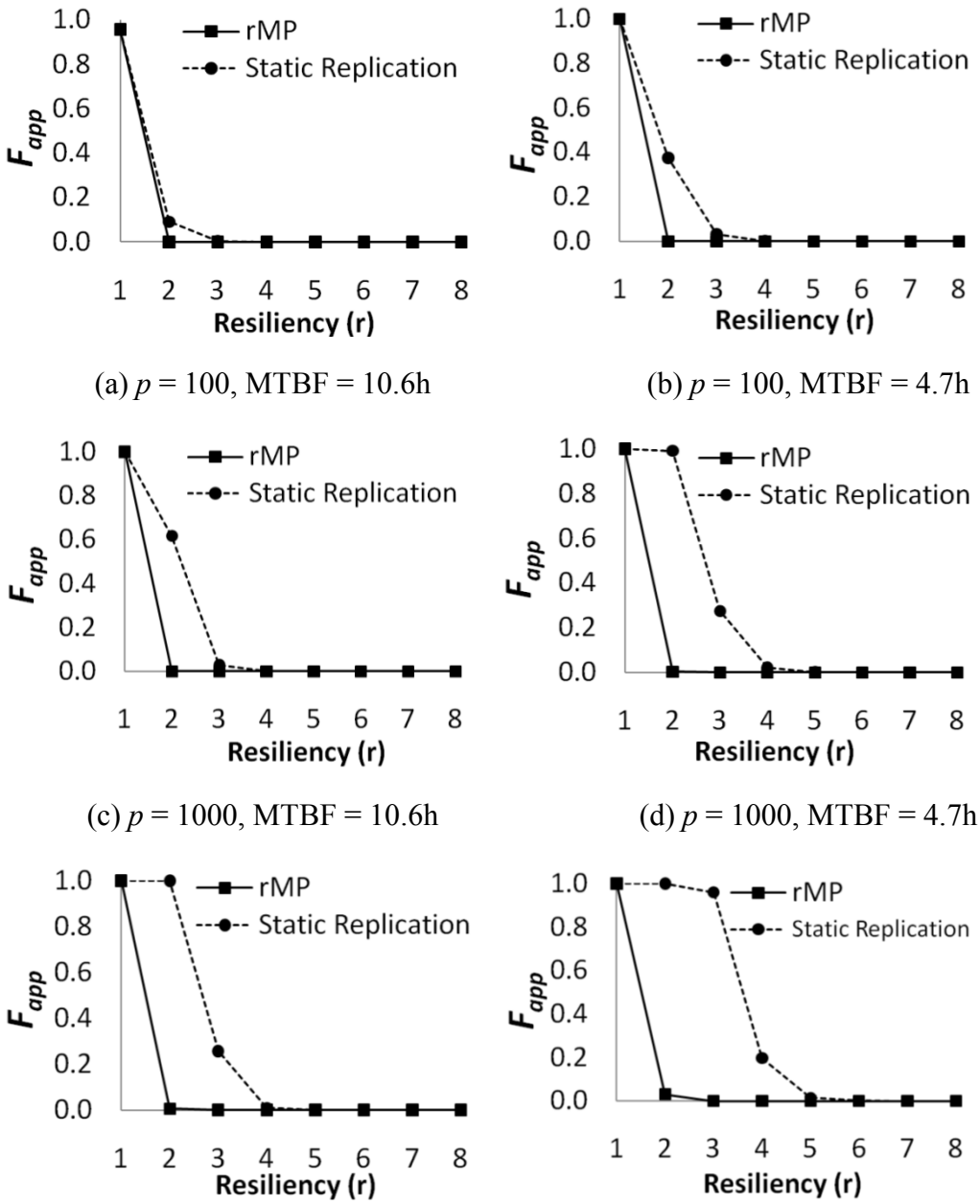
Platform	Type	Number of Processors	Years
LANL	HPC	4750	1996-2005
Grid5000	Grid	1288	2005-2006

A peak periods model of failures is used to characterize the failure history of these platforms [124]. The model parameters, presented in Table 6, describe the average duration of failure peaks and the system MTBF during these peaks. The average duration of failure peaks is greater than an hour for both systems, long enough to have a sustained impact on applications. In addition, the system MTBF is adjusted to an individual processor MTBF, using (17). The processor MTBF and failure rate are included in the table. This adjustment enables calculation of the probability of individual processor failures according to (16). This calculation assumes correlated faults *within* peak periods are independently distributed and obey an exponential failure distribution. While these assumptions are not necessarily valid in the case of correlated faults, they provide a first order approximation for this analysis. Like the independent fault model, this model assumes all faults are transient or may be repaired manually, and the outcome of a fault is complete process failure for the process that activates it when it occurs.

Table 6. Failure Parameters for Correlated Failure Analysis

Platform	No. of Processor	Avg. Failure Peak Duration (hr)	System MTBF (hr)	Processor MTBF (hr)	Processor λ (failures/yr)
LANL	4750	1.1	0.0022	10.6	826
Grid5000	1288	1.4	0.0036	4.7	1883

Figure 46 shows the probability of application failure due to complete process failures caused by correlated faults for rMP and static replication. The left column of the figure shows the response to the LANL model with the MTBF = 10.6 hours, and the right column shows the Grid5000 fault model with the MTBF = 4.7 hours. With correlated faults, the probability of application failure is orders of magnitude greater than with independent faults. All applications without replication have at least 0.95 probability of failure within 20 minutes of execution. This result demonstrates the necessity for fault tolerance in mission-critical applications on large scale architectures.



(a) p = 100, MTBF = 10.6h

(b) p = 100, MTBF = 4.7h

(c) p = 1000, MTBF = 10.6h

(d) p = 1000, MTBF = 4.7h

(e) p = 10,000, MTBF = 10.6h

(f) p = 10,000, MTBF = 4.7h

Figure 46. Probability of application failure due to complete process failures caused by correlated faults for rMP and static replication. The left collumn uses the LANL MTBF = 10.6h, and right column uses the Grid5000 MTBF = 4.7h.

The probability of failure for rMP applications is less than static replication for the same level of resiliency of all applications. Table 7 displays the required level of resiliency to maintain acceptable probability of failure for mission-critical applications. Static replication requires greater levels of resiliency than the rMP technology. In addition, the failure time interval and the required level of resiliency increase with the application service interval for applications using static replication but not for those using rMP.

Table 7. Level of Resiliency Required for Mission-critical Reliability ($F_{app} < 10^{-6}$)

Model	rMP	Static Replication
MTBF = 10.6 h, p = 100	3	6
MTBF = 10.6 h, p = 1000	3	6
MTBF = 10.6 h, p = 10,000	4	7
MTBF = 4.7 h, p = 100	3	7
MTBF = 4.7 h, p = 1000	4	8
MTBF = 4.7 h, p = 10,000	4	9

9.4.3 Computer Worm Model

A computer worm is a program that is able to self-replicate and propagate throughout a computer network without user intervention. In the last ten years, prolific worms have captured the attention of security researchers [125], [126], [127]. The Code Red worm, first released in 2001, targeted a Microsoft IIS web server vulnerability and propagated by scanning randomly generated IP addresses for the same vulnerability [126]. Subsequently, more efficient worms have emerged or have been theorized, such as Code Red II, Nimda, the Warhol worm, and Flash worms [127]. Two models are derived to determine the probability of application failure on a network penetrated by a computer

worm. Figure 47 provides a summary of the key assumptions, parameters, and equations of the computer worm analysis.

Computer Worm Model

Goal: Determine the probability of application failure on a network penetrated by a computer worm. Two models are considered: an unmitigated worm and a worm in a network with countermeasures in place.

Assumptions for Both Models:
All hosts are connected.
All hosts are vulnerable.
Vulnerabilities are in application software only. The operating system is reliable.
Host infections cause Byzantine process failures.

Unmitigated Worm Assumptions:
Host infections are permanent.

Worm with Countermeasures Assumptions:
Host infections are temporary due to a repair mechanism.
The repair mechanism restores infected applications to a gold standard.
Repairs occur at a constant repair rate of D repairs/hour.

Parameters:
N = 10,000 hosts in the network.
K = 2.6 compromises/hour
D = 1, 2, 2.5, 2.7, and 120 repairs/hour
t = time since initial infection
a = proportion of hosts infected in the network

Equations:
Probability of application failure from Byzantine process failures:

$$F_{app,B} = 1-(1-P_f^{(floor((r-1)/2+1))})^p \quad (15)$$

Proportion of hosts infected by an unmitigated worm as a function of time:

$$a = \frac{e^{K(t-T)}}{1+e^{K(t-T)}} \quad (20)$$

Proportion of hosts infected by a worm with countermeasures as a function of time:

$$a = (\frac{K-D}{K})\frac{e^{(K-D)(t-T)}}{1+e^{(K-D)(t-T)}} \quad (22)$$

Figure 47. Summary of assumptions, parameters, and equations for the analysis of the probability of application failure in the presence of two computer worm models.

The Random Constant Spread (RCS) model is a worm model developed by Staniford *et al*. [127] through analysis of the Code Red worm. In the RCS model, *N* is the number of vulnerable hosts on a network. For simplicity, it is assumed that all host are vulnerable (at the application level only) and that the network is a complete graph in which all hosts are connected [126]. Let *K* be the constant average compromise rate, or the average number of vulnerable machines that an infected host can compromise per hour. Define *a(t)* as the proportion of vulnerable hosts that have been compromised at time *t*, where *t* is measured from the initial infection. The number of infected hosts, *n(t)*, at time *t* is defined by (18).

$$n(t) = a(t) * N \tag{18}$$

Assuming that a host can only be infected once and stays infected thereafter, a simple differential equation for the system is defined by (19) with a solution defined by (20) [127].

$$\frac{da}{dt} = Ka(1 - a) \tag{19}$$

$$a = \frac{e^{K(t-T)}}{1+e^{K(t-T)}} \tag{20}$$

In (20), *T* is a constant of integration that fixes the time of the initial incident. This solution has been fitted to the data measured in several worm outbreaks to estimate values for *K*. Staniford *et al.* [127] proposed a model for a Code Red-like worm, capable of scanning 10 hosts/second using $K = 2.6$ compromises/hour. The compromise rate observed in the network is significantly lower than the scan rate because the worm generates random addresses within in the IP address space of size 2^{32}, and the overwhelming majority of addresses scanned are misses in the network.

The Code Red-like model can be used to model the spread of a random scanning worm in a network after an infection. Figure 48 displays the number of infected hosts as a function of time in a network of 10,000 hosts with K = 2.6 compromises/hour. This curve is obtained through a numerical model of (19) with one infection at time zero or from a plot of (20) with T = 3.54 hours, as determined by the initial conditions. In general, the shape of the curve is not affected by the number of hosts in the network, but the number of hosts and the initial conditions can shift the curve in time. Note that, unmitigated, the worm can infect the entire network within 6 hours. In this analysis, it is assumed that worm infections on the host cause *Byzantine* process failures for rMP applications.

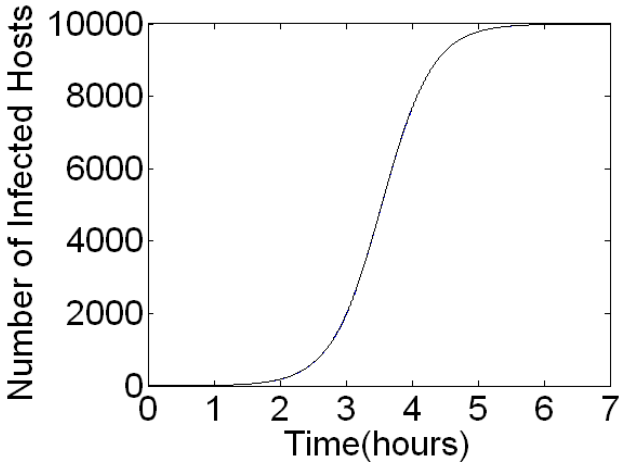


Figure 48. Number of infected hosts v. time of propagation of Code Red-like scanning worm in a network of 10,000 hosts.

The impact of a computer worm on a network is modeled under two different conditions. The first model, common in the literature, assumes that the worm propagates too quickly for response [125]. There are no countermeasures to mitigate the worm

propagation, and it is assumed that hosts are permanently infected. In this case, the RCS model is used to determine the probability of infection of a given host in the network as defined by the fraction of infected hosts in the network, according to (20). In a network of homogeneous hosts, this is also the probability of failure of a given process in the network. Figure 49 displays the probability of host infection as a function of time after the initial infection.

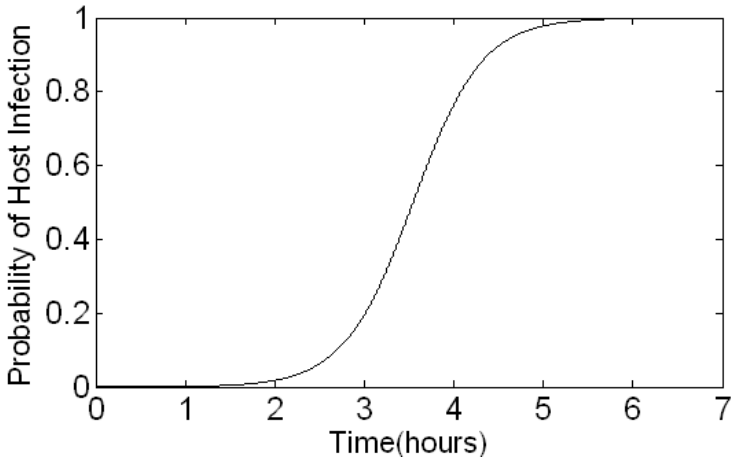


Figure 49. Probability of host infection v. time for the propagation of a Code Red-like scanning worm in a network of 10,000 hosts.

The second model of worm propagation includes countermeasures to slow the propagation of the worm. The model incorporates a repair mechanism capable of removing the worm infection from the host application. This repair mechanism may be an application regeneration, similar to the rMP technology, or an alternative technique provided by the host that restores applications from a gold standard. Regardless of the mechanism, it is assumed that the application is restored to a gold standard after infection. This system is modeled by modifying (19) to include the rate of repair for

infected hosts, D repairs/hour. The new model assumes a constant repair rate is applicable to infected hosts of the network [128] and is defined by (21) with a solution given by (22).

$$\frac{da}{dt} = Ka(1-a) - Da \tag{21}$$

$$a = \left(\frac{K-D}{K}\right)\frac{e^{(K-D)(t-T)}}{1+e^{(K-D)(t-T)}} \tag{22}$$

Equation (22) provides the probability of infection of a given host in the network with a constant repair rate.

Figure 50 displays the probability of host infection as a function of time after the initial infection for a range of repair rates. These repair rates include values that are less than, comparable to, and greater than the compromise rate, K. The slower repair rates, D = 1 and D = 2 repairs/hour demonstrate that countermeasures slow the propagation of the worm until the proportion of infected machines reaches a steady state value of $(K-D)/K$. The repair rates that are comparable to the compromise rate, D = 2.5 and 2.7 repairs/hour, appear to cause virtually zero probability of host failure. For D = 2.5 repairs/hour, this value converges to 0.038 probability of failure. However, for all values of D > K, the probability of host failure converges to zero. The repair rate of D = 120 repairs/hour is included to represent a mechanism similar to rMP process regeneration, in which failures are regenerated in 30 seconds.

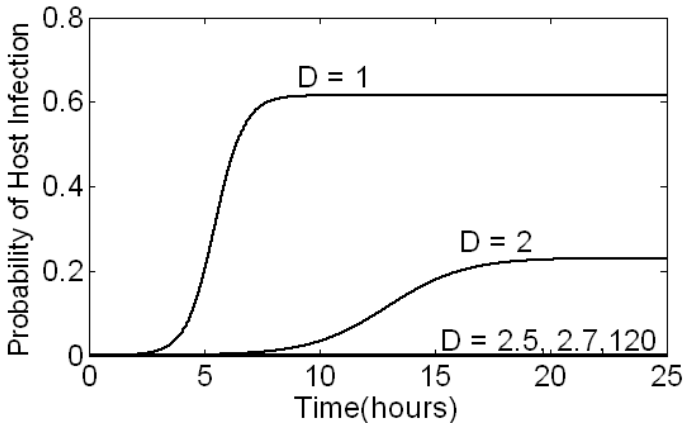
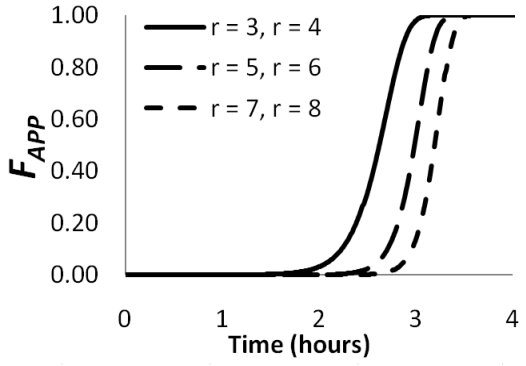


Figure 50. Probability of host infection v. time for the propagation of a Code Red-like scanning worm in a network of 10,000 hosts for rate of repair D = 1, 2, 2.5, 2.7, 120 repairs/hour.

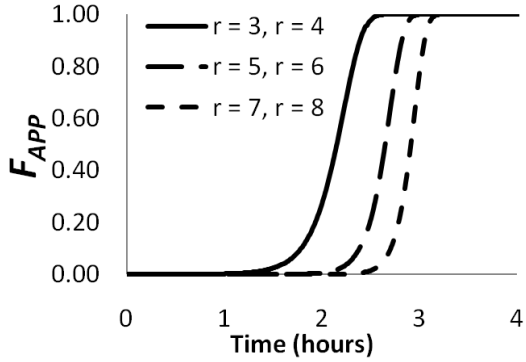
The reliability analysis of rMP applications under computer worm attack is considered with and without countermeasures. However, there is no distinction made between rMP and static replication technologies in the worm analysis. This is because the rMP technology has no knowledge of which hosts are infected at a given time and no mechanism to prevent regenerating a failed process on an infected host. As a result, the impact of process regeneration is reduced because processes are equally probable to fail on an infected host as they are to be regenerated on an infected host. At the same time, the model with countermeasures is considered for rMP applications exclusively. It is assumed that systems hosting static replication technologies have no mechanism to recover applications infected by the computer worm.

Figure 51 displays the probability of application failure as a function of time due to Byzantine process failures from an unmitigated worm propagation. Not surprising, the probability of application failure exceeds mission-critical levels within the first 1.8 hours

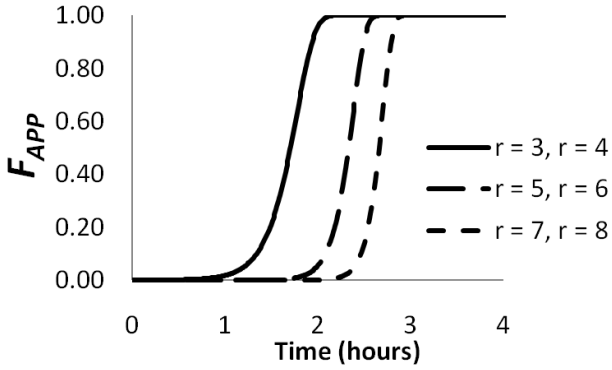
of the attack for all applications. Applications utilizing only triple resiliency exceed mission-critical levels in the first *minute* of the attack. While these results are discouraging, they represent a worst case scenario in worm propagation. Mission-critical networks are likely to deploy countermeasures to mitigate the propagation rate and would not fit this model.



(a)



(b)

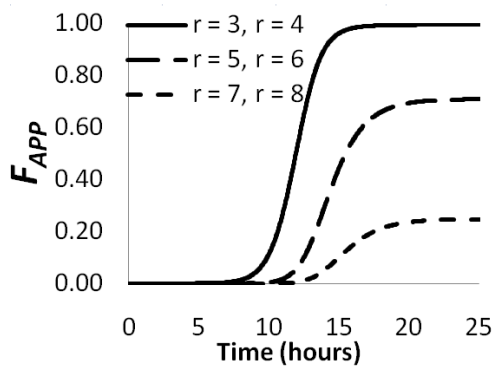


(c)

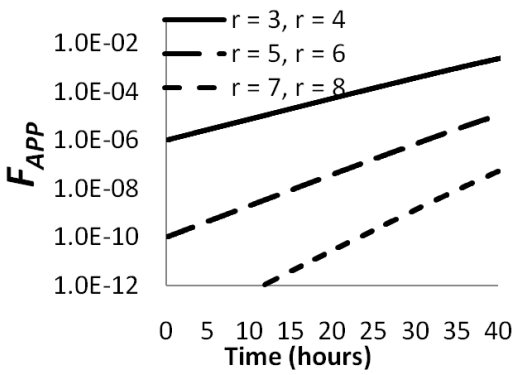
Figure 51. Probability of application failure as a function of time due to Byzantine process failures caused by an unmitigated worm attack for rMP replications with increasing levels of resiliency of (a) 100, (b) 1000, and (c) 10,000 processes.

Regardless of the propagation model, the overlapping curves displayed in Figure 51 demonstrate an important effect in Byzantine failure detection. For the purposes of majority voting, odd levels of resiliency are optimal. Two replicas are required for an increase in the number of failures allowed in a single process group. As a result, there is no gain in reliability from increasing to an even level of resiliency, as seen in the figure. However, there is a reduction in performance.

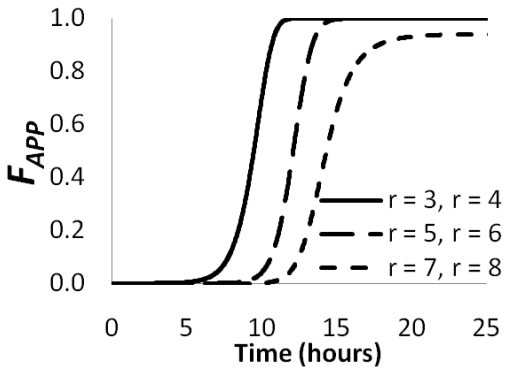
Figure 52 displays the probability of application failure as a function of time due to Byzantine process failures from a worm attack in the presence of countermeasures. The left column of the figure includes a repair rate slower than the compromise rate, D = 2 repairs/hour. The right column includes a repair rate comparable to the compromise rate, D = 2.5 repairs/hour. Note that the time scales in Figure 52 are greater than the scales in Figure 51. Both repair rates successfully slow the rate of worm propagation in the network. With a slow repair rate, the probability of application failure still exceeds mission-critical levels for all applications, but it takes longer for this to occur. Applications utilizing triple resiliency exceed mission-critical probabilities of failure within a minute, but applications with $r > 4$ maintain acceptable reliability for at least 2.6 hours. The reliability of rMP applications increases with a slow repair rate, but application failure is still imminent.



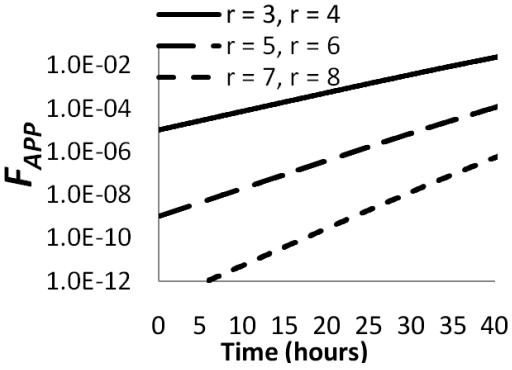
(a) p = 100, D = 2 repairs/hour



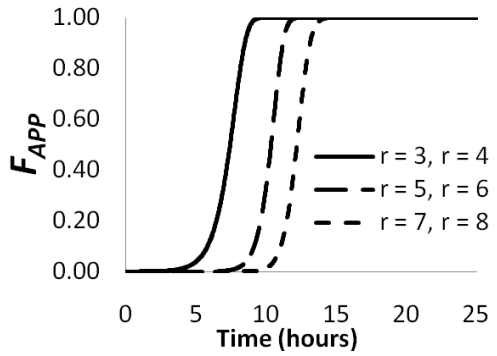
(b) p = 100, D = 2.5 repairs/hour



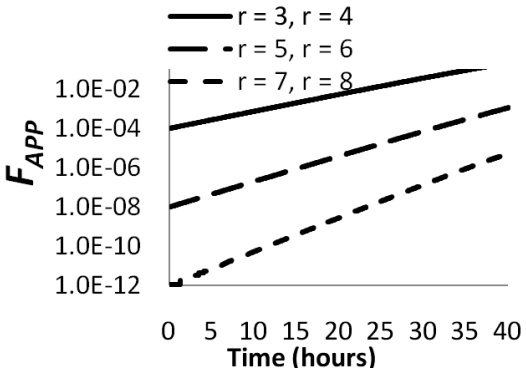
(c) p = 1000, D = 2 repairs/hour



(d) p = 1000, D = 2.5 repairs/hour



(e) p = 10,000, D = 2 repairs/hour



(f) p = 10,000, D = 2.5 repairs/hour

Figure 52. Probability of rMP application failure due to Byzantine process failures caused by worm propagation with repair mechanims. The left collumn uses a rate D = 2 repairs/hour, and right column uses a rate D = 2.5 repairs/hour.

The right column of Figure 52 displays the probability of application failure with comparable repair and compromise rates on a logarithmic scale. These probabilities are orders of magnitude less than the probabilities with a slower repair rate of D = 2 repairs/hour. While some applications quickly exceed mission-critical probabilities of failure, the applications with $r > 4$ levels of resilience have acceptable probabilities for the first 15 hours of the attack. This result represents significant improvement in application reliability, even with repair rates that are slightly slower than the compromise rate.

When the repair rate exceeds the compromise rate, the probability of application failure is zero; all rMP applications with $r > 2$ achieve mission-critical reliability. Because the rate of repair is greater than the rate of compromise, the proportion of infected machines always decreases after the initial infection. In this model, the initial infection penetrates a single host. Because all rMP applications with $r > 2$ are resilient to a single Byzantine process failure, they are able to operate through the attack.

9.5 Summary and Conclusions

This analysis explores the reliability of resilience in several fault and threat models. The models considered do not span the threat space of distributed applications, but they include a sample of likely failure scenarios. In general, resilience provides sufficient reliability to independent fault and correlated fault models. Relative to static replication technologies, resilience has a distinct advantage in reliability in the presence of correlated faults.

The reliability of rMP applications in the context of an unmitigated worm is unsatisfactory. This result highlights a critical aspect of resilience: no one technology

provides absolute reliability. This notion is well recognized in the mission-critical realm in which a collection of security technologies are applied as a comprehensive defense strategy. The value of this approach is demonstrated by the increase in reliability of rMP applications under computer worm attack when countermeasures are employed. When the countermeasures operate faster than the threat, rMP applications operate through attack to complete their mission. This result makes a strong case for measures that might slow the propagation of an attack, such as diversification techniques, or that quickly repair failed and infected components, such as regeneration. This outcome suggests that, provided rMP is used as a part of a comprehensive strategy, it achieves acceptable reliability for mission-critical applications.

On a final note, there is a surprising lack of concrete reliability analysis of resilient technologies in the literature. In most cases, analysis is limited to the assessment of scalable multiprocessor systems with independent and identically distributed faults. Unfortunately, in light of this analysis, the independent fault model is relatively insignificant for mission-critical applications. The correlated fault and computer worm models require a number of assumptions about the models themselves, the underlying technology, and the service requirements of the applications. Perhaps the uncertainty in these assumptions prevents investigators from exploring quantitative metrics. However, even with limited confidence in the underlying assumptions, this analysis provides worthwhile insights into the relative reliability of alternative replication strategies.

Chapter 10 Summary and Conclusions

This thesis describes operating system *mechanisms* and *policies* to allow distributed applications to operate through computer failures and attacks. The rMP technology provides mechanisms to dynamically replicate processes, detect inconsistencies in their behavior, and restore the level of assurance as a computation proceeds. A resilient process management strategy is devised to schedule resilient processes of rMP applications. By integrating properties of diffusion and robotic swarming algorithms, the DIFFUSE algorithm shows potential to meet the requirements of load-balancing performance and resilience in a single strategy. Together, these technologies represent a central step toward automated, transparent, and scalable resilience for distributed applications.

10.1 Limitations and Future Recommendations

The limitations of the rMP technology that warrant further exploration are addressed with the hope that lessons learned in this thesis will benefit the development of future technologies.

10.1.1 Linux-based Resilience

The rMP technology is built on top of the Linux kernel. The commitment to a kernel-based technology enables transparency and versatility in the resiliency mechanisms. Unfortunately, this decision prevents portability. In its current form, rMP is not portable to other operating systems, or even some previous versions of Linux. Linux was selected because it has several powerful built-in features to support the investigation of advanced algorithms. This kernel support, coupled with prior work in

Linux-based fault tolerance provides an easy development path to explore novel concepts for resilience.

However, there are two downsides to Linux-based resilience. The first is that it limits the deployment potential of the prototype. This limitation is not a problem if the target platforms are HPC systems. As of June 2011, Linux is used in 91% of the top 500 supercomputers [129]. However, there is an increasing trend in both industry and government toward cloud computing. Cloud computing is characterized by diversity in platform architectures and software [130]. Thus, the current rMP technology is not well situated to follow the cloud computing trend.

The second downside to Linux-based resilience is that platform presents a homogenous attack surface. If every host of an rMP application is the same, the vulnerabilities are also the same. As a result, a single software bug or computer attack may compromise every host, as seen in the reliability analysis of a computer worm attack. In this event, the value of replication is reduced. In fact, this shared vulnerability may work to an attacker's advantage by creating a replicated point of entry to take root in the system. To mitigate this effect, diversity may be integrated at various levels of the software stack to alter the attack surface [131].

10.1.2 Locality and Resilience

Process locality is considered an asset in traditional resource management due to its performance benefits [76]. However, to the extent that computer faults and attacks are spatially correlated, locality may undermine resilience. Some argue that geographically distributed replication is the best strategy for reliability [132]. This logic is sound in the context of faults which are the result of the physical environmental conditions,

catastrophic events, or localized power outages. However, because locality bolsters failure detection in the rMP approach, these effects may be mitigated by the detection and regeneration of failed processes.

The tradeoffs in locality and resilience warrant further investigation. A more thorough reliability analysis that includes both spatial and temporal relationships is recommended. To be useful, this analysis also requires corresponding fault and threat models with the same information. Unfortunately, these models represent a significant research undertaking, but they would enable more sophisticated analysis of reliability and a better understanding of resilient strategies.

10.1.3 Non-determinism and Resilience

The proposed solution to non-determinism in the rMP technology is a stop-gap to enable resilience for existing applications that utilize wildcard operations. The drawbacks with respect to code management and runtime performance are considerable. However, the greater issue is that the solution only addresses explicit non-determinism that is a result of wildcard messaging. There are other sources of non-determinism in applications that are problematic for resilience. Consider any program that utilizes random or probabilistic operations. For example, in scientific computing a manager-worker communication scheme may be used to parallelize a set of stochastic simulations [34]-[35]. These simulations are inherently and purposely non-deterministic, and replicated simulations would not be consistent. This problem suggests that a more flexible solution is required to provide resilience for a larger variety of applications.

Alternative strategies to preserve resilience and non-determinism are allocated to future work. One approach may be to extend the notion of coordination of process

groups to enforce consistency in the general case. In this approach, diverging replicas are identified through communication or any other means. One replica state is selected, and all inconsistent replicas are killed and regenerated from the selected state. This selection may be based on majority voting, if a minimum number of replicas are consistent, or it may be arbitrary. Obviously, this concept represents an extreme process regeneration policy for non-deterministic applications. Another approach may be to extend the notion of dynamic process replication for non-deterministic applications. The extension would allow diverging replicas to be identified and process regeneration to be used to restore the level of resiliency *for each replica*. An obvious problem with this approach is the potential for exponential increase in the number of replicas in a system. At this point, these approaches are merely suggestions for exploration in the problem space.

10.1.4 Obscured Process Failures

The rMP technology relies on detecting process failures through group communication. This reliance requires that process failures create communications that deviate from expectation. In many applications, this is true. Processes that experience complete failure will cease communications and be detected through communication timeouts. However, Byzantine process failures are less cooperative. The strategy of resilience assumes that the content of messages is an indicator of comprehensive process health. In many applications, this assumption is true because the most critical pieces of data are shared with other processes—the result of a computation or a required boundary condition. However, there are exceptions to this rule. Consider an application in which processes store critical output in a file rather than communicating it to other processes. Process inconsistencies are not detected in this case.

To remedy this problem, the rMP technology may expand its failure detection protocols to include a more comprehensive set of output channels, such as the file system, shared memory areas, and arbitrary I/O devices. Alternatively, a fingerprinting approach may be used in which the entire process state of each replicated process is hashed and the hashes compared for consistency. This method represents a more comprehensive approach to failure detection but would require services external to the current message-passing technology.

10.1.5 Computer Security and the Intelligent Attacker

This thesis attempts to differentiate between the fault tolerant and computer security communities throughout its discussion and analysis. Many researchers regard the two perspectives as the same. However, there are crucial distinctions, and the rMP technology provides a clarifying example. The reliability analysis of resiliency is conducted with respect to the application. The reliability of the rMP technology *itself* and the operating system of the host is presumed. This thinking is consistent with the fault tolerant perspective.

The computer security perspective is more likely to emphasize the underlying technology to consider how an intelligent attacker would compromise rMP applications. This analysis would target vulnerabilities in the implementations of the communication and migration modules of the rMP technology. Unfortunately, the current prototype would not stand up to rigorous security analysis. Because the prototype is a demonstration-of-concept, some security measures are simply earmarked for future work. Encryption technologies are planned for application and control messages to provide the major security tenets—confidentiality, authenticity, integrity, and non-repudiation. More

comprehensive input sanitization, error checking, status acknowledgements, and control redundancies are intended to provide robustness. However, these may not be enough. There are threat vectors available to the creative attacker that require further investigation. The technology, like any software object, requires further analysis and penetration testing to properly evaluate its security posture.

10.2 Broader Implications

10.2.1 Swarm Inspired Process Management

Swarm-inspired process management is relatively new and may offer alternative solutions to a wide range of problems in distributed resource management. As multi-core computers become ubiquitous, an increase in number and diversity of devices and communication networks available as computational engines is evident. Distributed systems are now interconnected through a wide variety of local area, wide area, and wireless networking technologies. This diversity confounds the problem of distributed resource management and the notion of *local* communication.

The process management approaches described in this thesis include a spectrum of communication ranges for distributed management. The BRW algorithm is an autonomous algorithm with no communication. This autonomy could be applied to any computer architecture. The original heat diffusion algorithm conducts only nearest neighbor communication. This is a natural communication scheme for large scale architectures with regular grid connectivity, such as mesh architecture. The GSO and DIFFUSE algorithms represent an intermediate communication class between nearest neighbor and global communication featuring limited range communication. These algorithms may be applicable to a wider variety of communications networks. For

example, in local area networks, locality may be defined by the number of network hops whereas in wireless networks, locality may be defined by a physical distance. Further, limited range communication algorithms may provide the potential to tune the communication range to a specific platform or network. This versatility is a potential advantage of the DIFFUSE algorithm and other algorithms in this class.

10.2.2 Next Generation Operating Systems

These technologies form the basis for application resilience in next generation operating systems. The prototypes developed to date serve as a demonstration of resiliency concepts, but Linux is unlikely to be useful as the foundation of a resilient system. This research serves as a setting to explore alternative algorithms and strategies to be integrated in a clean-slate approach. Many insights will be applied or explored in next generation designs, some of which have already been discussed:

- Integrating swarm-inspired algorithms with the operating system scheduler
- Optimizations of the message transport mechanisms
- Tuning adaptive failure detection
- Securing the operating system technologies to malicious attacks
- Exploring advance solutions for non-deterministic applications

The next generation design includes data structures, communication protocols, memory systems, and distributed schedulers to enable dynamic process replication, migration, regeneration, and mapping mechanisms that are simple, efficient, and scalable [131].

Summary of Publications

Journal Publications:

- K. McGill and S. Taylor, "Robot Algorithms for Localization of Multiple Emission Sources," ACM *Computing Surveys (CSUR)*, vol. 43, no. 3, 2011.

Peer-Reviewed Conference Publications:

- K. McGill and S. Taylor, "Computational Resiliency for Distributed Applications," *MILCOM 2011*, Baltimore, MD, Nov 2011, accepted for publication.
- K. McGill and S. Taylor, "Application Resilience with Process Failures," *Proc. of 2011 Int. Conf. on Security and Management*, Las Vegas, NV, July 2011.
- K. McGill and S. Taylor. "DIFFUSE algorithm for robotic multi-source localization", *Proc. of IEEE Int. Conf. on Practical Robot Applications (TePRA)*, April 2011.
- K. McGill and S. Taylor, "Comparing swarm algorithms for large scale multi-source localization," In *Proc. of TePRA*, Woburn, Massachusetts, Nov 2009.
- K. McGill and S. Taylor, "Comparing swarm algorithms for multi-source localization," *Proc. of IEEE Int. Wkshp. on Safety, Security, and Rescue Robotics*, Denver, Colorado, Nov 2009.

Submissions with Decisions Pending:

- K. McGill and S. Taylor, "Operating System Support for Resilience," *IEEE Transactions on Reliability*, submitted for publication.
- S. Taylor, M. Henson, M. Kanter, S. Kuhn, K. McGill, and C. Nichols, "Bear—A Resilient Operating System for Scalable Multi-processors," *IEEE Security and Privacy*, Nov/Dec 2011, submitted for publication.

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] H. Debar, M. Dacier, and A. Wespi. "A revised taxonomy for intrusion-detection systems," *Annals of Telecommunications*, vol. 55, no. 7, pp. 361-378, 2000.

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

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

[6] 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.

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

[8] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2003, pp. 191-206.

[9] 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.

[10] N.A.Quynh and Y. Takefuji, “Towards a tamper-resistant kernel rootkit detector,” in *Proceedings of the 2007 ACM Symposium on Applied Computing*, 2007, pp. 276-283.

[11] 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.

[12] H. Zhong and J. 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] G. Zheng, C. Huang, L.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.

[15] 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.

[16] 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.

[17] D.S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler and S. Zhou. "Process migration," *ACM Comput. Surv, vol. 32, no. 3*, pages 241-299, 2000.

[18] 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.

[19] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software- Practice and Experience,* vol. 2, no. 8, pages 757-785, 1991.

[20] G. Valle, C. Morin, J. Berthou, I. Dutka Malen, and R. Lottiaux, "Process migration based on gobelins distributed shared memory," in *Proceedings of the workshop on Distributed Shared Memory*, pages 325-330, May 2002.

[21] C. Wang, F. Mueller, C. Engelmann, and S. Scott, "Proactive Process-Level Live Migration in HPC Environments," in *Proceedings of the 2008 ACM/IEE Conference on Supercomputing*, 2008.

[22] G. Janakiraman, J. Santos, D. Subhraveti, and Y. Turner, "Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, pages 260-269, 2005.

[23] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," in *Proceedings of the 5th*

Symposium on Operating Systems Design and Implementation (OSDI), pages 361-376, 2002.

[24] 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.

[25] 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.

[26] R. Brightwell, K. Ferreira, and R. Riesen, "Transparent Redundant Computing with MPI," in *Proceedings of EuroPVM/MPI*, 2010.

[27] 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.

[28] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhua, A. Skjellum, Y. Dandass, and M. Apte, "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).

[29] 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.

[30] J. Lee and S. Taylor, "Advances in Computational Resiliency," in *Proceedings of the IEEE Aerospace Conference*, Big Sky, Montana, March 2001.

[31] K. McGill and S. Taylor, “Application Resilience with Process Failures,” *The 2011 International Conference on Security and Management*, Las Vegas, NV, July 2011.

[32] K. McGill and S. Taylor, “Computational Resiliency for Distributed Applications,” *MILCOM 2011*, submitted for publication.

[33] K. McGill and S. Taylor, “Operating System Support for Resilience,” *IEEE Transactions on Reliability*, submitted for publication.

[34] K. McGill and S. Taylor, “Comparing swarm algorithms for multi-source localization,” In *Proceedings of the 2009 IEEE International Workshop on Safety, Security, and Rescue Robotics*, Denver, Colorado, November 3-6, 2009.

[35] K. McGill and S. Taylor, “Comparing swarm algorithms for large scale multi-source localization,” In *Proceedings of the 2009 IEEE International Conference on Technologies for Practical Robot Applications*, Woburn, Massachusetts, November 9-10, 2009.

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

[37] G. Cybenko, “Dynamic load balancing for distributed memory multiprocessors,” *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279-301, 1989.

[38] Heirich and S. Taylor "Load Balancing by Diffusion", in *Proceedings of 24th International Conference on Parallel Programming*, 1995, pp. 192-202.

[39] J. Sterbenz, D. Hutchinson, and E. K. Cetinkaya, A. Jabbar, J.P. Rohrer, M. Scholler, and P. Smith "Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines," *Computer Network: Special Issue on Resilient and Survivable Networsk*, vol. 54, no. 9, pp. 1245-1265, June 2010.

[40] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, March 2004.

[41] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem" *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp.382–401, July 1982.

[42] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[43] R. Lottiaux and C. Morin, "Containers: A sound basis for a true single system image," *In Proceeding of IEEE International Symposium on Cluster Computing and the Grid*, pages 66-73, May 2001.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

[55] M. K. Reiter, "The rampart toolkit for building high-integrity services," *In Theory and Practice in Distributed Systems*, vol. 938, pages 99–110. Springer-Verlag, Berlin Germany, 1995.

[56] M. Treaster, "A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems," *ACM Computing Research Repository* , 2005.

[57] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial, " *ACM Computing Surveys*, vol. 22, no. 4, pp. 299-319, Dec. 1990.

[58] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *In Proceedings of the Second, ACM SIGACT-SIGMOD Symposium on Principles of Database System (PODS)*, pp 1–7, 1983.

[59] R. Canneti and T. Rabin. "Optimal Asynchronous Byzantine Agreement," Technical Report no. 92-15, Computer Science Department, Hebrew University, 1992.

[60] M. Reiter, "A secure group membership protocol," *IEEE Trans. Softw. Eng.*, vol. 22, no. 1, pp 31-42, Jan 1996.

[61] J. Garay, and Y. Moses, "Fully polynomial Byzantine agreement in t + 1 rounds," *Proceedings of the 25th ACM symposium on Theory of computing*, San Diego, CA, May 1993.

[62] G. Bracha and S. Toueg, "Asynchronous Consensus and Broadcast Protocols," *Journal of the ACM*, vol. 32, no. 4, 1995.

[63] M. Castro and B. Liskov, “Practical Byzantine fault tolerance and proactive recovery,” *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[64] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, J. Wylie, “Fault-scalable Byzantine Fault-Tolerant Services,” *Association for Computing Machinery Symposium on Operating Systems Principles*, 2005.

[65] J. Cowling, Danial Myers, B. Liskov, R. Rodrigues, and L. Shrira, “HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance,” *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[66] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, E. Wong, Zyzzyva, “Speculative Byzantine Fault Tolerance,” *ACM Transactions on Computer Systems*, vol. 27 no. 4, December 2009.

[67] Message Passing Interface Forum, “MPI: A Message Passing Interface,” *In Proc. of Supercomputing ’93*, pages 878–883, November 1993.

[68] I. Foster. *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[69] S. Taylor and J. 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.

[70] R. Love, Linux Kernel Development, 2nd ed., Novell Press, 2005.

[71] F. Chang, J. Dean, S. Ghenawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, “Bigtable: A Distributed Storage System for Structured

Data," in *Proceedins of the 2006 Operating System Design and Implementation*, Seattle, WA, 2006.

[72] C. Nichols, S. Taylor, J. Keranen, and G. Schultz, "A Concurrent Algorithm for Real-Time Tactical LiDAR," *2011 IEEE Aerospace Conference*, 2011.

[73] M.E. Palmer, B. Totty, and S. Taylor, "Ray Casting on Shared-Memory Architectures: Efficient Exploitation of the Memory Hierarchy," *IEEE Concurrency,* vol. 6, no. 1, pages 20-36, 1998.

[74] M. Rieffel, M. Ivanov, S. Shankar, and S. Taylor, "Concurrent Simulation of Neutral Flow in the GEC Reference Cell," *Journal of Concurrency: Practice and Experience*, vol. 12, no. 1, pages 1-19, 2000.

[75] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279-301, 1989.

[76] Heirich and S. Taylor "Load Balancing by Diffusion", in *Proceedings of 24th International Conference on Parallel Programming*, 1995, pp. 192-202.

[77] A.B. Heirich, "Analysis of scalable algorithms for dynamic load balancing and mapping with application to photo-realistic rendering," Ph.D. dissertation, California Institute of Technology, Pasadena, CA, 1998.

[78] J. E. Boillat, "Load balancing and poisson equation in a graph," *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 289-313, 1990.

[79] P. Berenbrink, T. Friedetzky, and Z. Hu, "A new analytical method for parallel, diffusion-type load balancing," *Journal of Parallel and Distributed Computing,*vol. 69, pp. 54-61, 2009.

[80] E. Jeannot and F. Vernier, “A practical approach of diffusive load balancing algorithms,” in *Euro-Par 2006 Parallel Processing*, W. Nagel, W. Walter, and W. Lehner, Eds., Springer Verlag, 2006.

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

[82] Xu, B. Monien, R. Luling, and F.C.M.Lau, “Nearest neighbor algorithms for load balancing in parallel computers” *Concurrency: Practice and Experience*, vol. 7, no. 7, pp. 707-736, 1995.

[83] S.S. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan, “Analysis of a graph coloring based distributed load balancing algorithm,” *Journal of Parallel and Distributed Computing*, vol. 10, no. 2, pp. 160-166, 1990.

[84] M. Houle, A. Symvonis, and D.R. Wood, “Dimension-exchange algorithms for load balancing on trees,” in *Proceedings of the 9th International Colloquium on Structure Information and Communication Complexity*, 2002, pp. 181-196.

[85] G. Bronevich and W. Meyer, “Load balancing algorithms based on gradient methods and their analysis through algebraic graph theory,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 2, pp. 209-220, 2008.

[86] F.C.H. Lin and R.M. Keller, “The gradient model load balancing method,” *IEEE Transactions on Software Engineering*, vol. 13, no. 1, pp. 32-28, 1987.

[87] K. McGill and S. Taylor, “Robot Algorithms for Localization of Multiple Emission Sources,” ACM *Computing Surveys (CSUR)*, vol. 43, no. 3, 2011.

[88] Dhariwal, G.S. Suhkatme, and A.A.G. Requicha, "Bacterium-inspired robots for environmental monitoring," in *Proceedings of the 2004 International conference on Robotics & Automation*, New Orleans, Louisiana, April 2004, pp. 1436-1443.

[89] K.N. Krishnanand and D. Ghose, "A glowworm swarm optimization based multi-robot system for signal source localization," in *Design and Control of Intelligent Robotic Systems,* D. Liu, L. Wang, and K.C. Tan, Eds., Berlin, Germany, Springer Verlag, 2009, pp. 49-68.

[90] K.N. Krishnanand and D. Ghose, "Glowworm swarm optimization for simultaneous capture of multiple local optima of multimodal functions," *Swarm Intelligence*, vol.3, no. 2, pp.87-124, 2009.

[91] K.N. Krishnanand and D. Ghose, "Glowworm swarm based optimization algorithm for multimodal functions with collective robotics applications," *Multiagent and Grid Systems,* vol. 2, no. 3, pp.209-222, 2006.

[92] K.N. Krishnanand and D. Ghose, "Glowworm swarm optimization: A swarm intelligence based multimodal function optimization technique with applications to multiple signal source localization," Technical Report GCDSL 2007/05, Department of Aerospace Engineering, Indian Institute of Science, October 2007.

[93] X. Cui, C.T. Hardin, R.K. Ragade, and A.S. Elmaghraby, "A swarm approach for emission sources localization," in *Proceedings of the 16th International Conference on Tools with Artificial Intelligence*, Boca Raton, Florida, Nov 2004, pp. 424-430.

[94] X. Cui, R.K. Ragade, and A.S. Elmaghraby, "A collaborative search and engage strategy for multiple mobile agents with local communication in large scale hostile

area," *In Proceedings of the International Symposium on Collaborative Technologies and Systems*, San Diego, California, 244 – 249, January 2004.

[95] V. Gazi and K. Passino, "Stability analysis of social foraging swarms," *IEEE Transactions on Systems, Man, and Cybernetics—Part B: cybernetics,* vol. 34, no. 1, pp. 539-557, 2004.

[96] L. Marques, U. Nunes, and A.T. De Almeida, "Odour searching with autonomous mobile robots: an evolutionary-based approach," in *Proceedings of the IEEE Int. Conf. on Advanced Robotics*, Combra, Portugal, June 2003, pp. 494-500.

[97] L. Marques, U. Nunes, and A.T. De Almeida, "Particle swarm-based olfactory guided search," *Autonomous Robots,* vol. 20, pp. 277-287, 2006.

[98] J. Pugh, and A. Martinoli, "Distributed Adaptation in multi-robot search using particle swarm optimization," in *Proceedings of the 10th International Conference on the Simulation of Adaptive Behavior*, Osaka, Japan, July 2008, pp. 393- 402.

[99] P. Scerri, T. Von Gonten, G. Fudge, S. Owens, and K. Sycara SCERRI, P., "Transitioning multiagent technology to UAV applications," in *Proceedings of 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, Estoril, Portugal, May 2008, pp. 89-96.

[100] K.M. Sim and W. H. Sun, "Ant colony optimization for routing and load-balancing: Survey and new directions," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans,* vol. 33, no. 5, pp. 560-572, 2003.

[101] H. Chen, L. Ping, X. Pan, K. Lu and X. Jiang, "A swarm-inspired resource distribution for SMT processors," *Presented at BIONETICS '08: Proceedings of the*

3rd International Conference on Bio-Inspired Models of Network, Information and Computing Sytems, Hyogo, Japan, Nov 2008.

[102] A. Moallem and S. A. Ludwig, "Using artificial life techniques for distributed grid job scheduling," *Proceedings of the 2009 ACM Symposium on Applied Computing,* Honolulu, Hawaii, March 2009.

[103] A. Ali, M. A. Belal, and M.B. Al-Zoubi, "Load Balancing of Distributed Systems Based on Multiple Ant Colonies Optimization," *American Journal of Applied Sciences,*vol. 7, no. 3, pp. 433-438, 2010.

[104] J. Kennedy, and R.C. Eberhart, "Particle swarm optimization," *In Proceedings of the IEEE International Conference on Neural Networks*, Perth, Australia, pp. 1942-1948, Dec 1995.

[105] A. Howard, M. Mataric, and G. Sukhatme, "Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem," in *Proceedings of the 6th International symposium on Distributed Autonomous Robotics Systems (DARSO2),* Fukuoka, Japan, June 2002, pp. 299-208.

[106] J. Cortes, S. Martinez, T. Karatax, and F. Bullo, "Coverage control for mobile sensing networks," *IEEE Transaction on Robotics and Automation*, vol. 20, no. 2, pp. 243- 255, April 2004.

[107] P. Ogren, E. Fiorelli, and N.E. Leonard, "Cooperative control of mobile sensor networks: adaptive gradient climbing in a distributed environment," *IEEE Transactions on Automatic Control*, vol. 49, no. 8, pp. 1292-1302, August 2004.

[108] W. Spears, D. Spears, J. Hamann, and R. Heil, "Distributed, physics-based control of swarms of vehicles," *Autonomous Robots*, vol. 17, no. 2-3, pp. 137-162, September 2004.

[109] B. Bhargava and L. Lilien, "Vulnerabilities and Threats in Distributed Systems," *Intl. Conf. on Distributed Computing & Internet Technology (ICDCIT 2004)*, Bhubaneswar, India, Dec. 2004, pp. 146-157.

[110] K.S. Trivedi, D. Kim, A. Roy, and D. Medhi, "Dependability and Security Models," *Proceedings of 7th International Workshop on the Design of Reliable Communication Networks (DRCN 2009)*, Washington, DC, October 2009.

[111] D. M. Nicol, W. H. Sanders, K. S. Trivedi, "Model-Based Evaluation: From Dependability to Security," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 1, 2004.

[112] R. J. Ellison, D. A. Fisher, R. C. Linger, H. F. Lipson, T. Longstaff, N. R. Mead, "Survivable Network Systems: An Emerging Discipline," Technical Report CMU/SEI-97-TR-013, 1997.

[113] P. E. Heegard, K. S. Trivedi, "Network survivability modeling," Computer Networks, vol. 53, no. 8, 2009.

[114] H. F. Lipson, D. A. Fisher, "Survivability—a new technical and business perspective on security," *Proc. of NSPW*, 1999.

[115] Y. Liu and K. S. Trivedi, "Survivability Quantification: The Analytical Modeling Approach," *Int. J. Performability Engineering*, vol. 2, no. 1, 2006.

[116] ANSI T1A1.2 Working Group on Network Survivability Performance, Technical Report on Enhanced Network Survivability Performance, ANSI, Tech. Rep. TR No. 68, 2001.

[117] E. Tran and P. Koopman, "Mission Failure Probability Calculations for Critical Function Mechanizations in the Automated Highway System," Carnegie Mellon University Robotics Institute Technical Report CMU-RI-TR-97-44, 1997.

[118] R. Reisen, K. Ferreira, and J. Stearley, "See Applications Run and Throughput Jump: The Case for Redundant Computing in HPC," In *Proceedings of the International Conference on Dependable Systems and Networks*, Chicago, IL, July 2010.

[119] E. N. Elnozahy, J. S. Plank, W. K. Fuchs, "Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Roll-back-Recovery," *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 2, 2004.

[120] D. P. Siewiorek, R. S. Swarz, Reliable Computer Systems: Design and Evaluation, 2nd ed., Digital Press, 1992.

[121] T. Courtney et al., "The Möbius Modeling Environment," *Tools of the 2003 Illinois Int'l Multiconference on Measurement, Modelling, and Evaluation of Computer Communication Systems*, Universität Dortmund Fachbereich Informatik research report no. 781, 2003.

[122] D. Tang, R. K. Iyer, "Analysis and Modeling of Correlated Failures in Multicomputer Systems," *IEEE Trans. on Computers*, vol. 41, no. 5, 1992.

[123] B. Schroeder, G. Gibson, "A Large-scale Study of Failures in High-performance-computing Systems," in *Proceedings of the International Conference on*

Dependable Systems and Networks (DSN2006), Philadelphia, PA, USA, June 25-28, 2006.

[124] N. Yigitbasi, M. Gallet, D. Kondo, A. Iosup, and D. Epema "Analysis and Modeling of Time-Correlated Failures in Large-Scale Distributed Systems," Delft University of Technology Parallel and Distributed Systems Report Series, no PDS-2010-004, 2010.

[125] G. Serazzi and S. Zanero, "Computer Virus Propagation Models," In *Tutorials of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS'03*), 2003.

[126] D. Moore, C. Shannon, and J. Brown, "Code-Red: a case study on the spread and victims of an Internet worm," *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, New York, New York, 2002.

[127] S. Staniford, V. Paxson, and N. Weaver, "How to own the internet in your spare time," in *Proceedings of the 11th USENIX Security Symposium*, 2002.

[128] Zou, C.C., Gong, W., Towsley, D. "Code red worm propagation modeling and analysis," in *Proceedings of the 9th ACM conference on Computer and communications security*, 2002.

[129] top500.org, "Operating System Family Share for 6/2011," http://top500.org/stats/list/37/osfam, June 2011 [July, 8 2011].

[130] National Institute of Standards, "The NIST Definition of Cloud Computing," NIST Special Publication 800-145, January 2011.

[131] S. Taylor, M. Henson, M. Kanter, S. Kuhn, K. McGill, and C. Nichols, "Bear—A Resilient Operating System for Scalable Multi-processors," *IEEE Security and Privacy*, Nov/Dec 2011, submitted for publication.

[132] D. Kurzyniec and V. Sunderam, "Failure resilient heterogeneous parallel computing across multidomain clusters," *International Journal of High Performance Computing Applications*, vol. 19, no. 2, pp. 143-155, 2005.

Appendix 1: Standard Benchmark Case Definitions

All three standard benchmark cases use the same gradient field consisting of ten Gaussian load troughs. The field strength at any point (x, y) in the field is a summation of the components from the individual load troughs given by (23).

$$f(x,y) = \sum_{i=1}^{10} I_i e^{-\frac{((x-x_i)^2+(y-y_i)^2)}{2\sigma_i^2}} \tag{23}$$

In (23), the ith trough is located at point (x_i, y_i) in the space and has intensity I_i and "width" σ_i. After calculating the total field strength from all troughs, any field strength less than a threshold of 0.5 is set to zero. The parameters for all load troughs are shown in Table 8.

Table 8. Standard Benchmark Field Parameters

Source i	x_i	y_i	I_i	σ_i
1	100	650	25	35
2	100	350	5	35
3	300	800	10	35
4	300	500	10	35
5	300	200	10	35
6	700	800	10	75
7	700	500	10	75
8	700	200	10	75
9	900	650	25	75
10	900	350	5	75

The initial process distributions for each benchmark case are the uniform, point, and line distributions. Table 9 displays the x and y coordinates for the initial positions of 100 processes in each distribution.

Table 9. Initial Process Positions for Uniform, Point, and Line Distributions

	Uniform		Point		Line	
Process	**x**	**y**	**x**	**y**	**x**	**y**
0	90.91	90.91	495.91	495.91	9.90	990
1	181.82	90.91	496.82	495.91	19.80	990
2	272.73	90.91	497.73	495.91	29.70	990
3	363.64	90.91	498.64	495.91	39.60	990
4	454.55	90.91	499.55	495.91	49.51	990
5	545.46	90.91	500.46	495.91	59.41	990
6	636.36	90.91	501.36	495.91	69.31	990
7	727.27	90.91	502.27	495.91	79.21	990
8	818.18	90.91	503.18	495.91	89.11	990
9	909.09	90.91	504.09	495.91	99.01	990
10	90.91	181.82	495.91	496.82	108.91	990
11	181.82	181.82	496.82	496.82	118.81	990
12	272.73	181.82	497.73	496.82	128.71	990
13	363.64	181.82	498.64	496.82	138.61	990
14	454.55	181.82	499.55	496.82	148.52	990
15	545.46	181.82	500.46	496.82	158.42	990
16	636.36	181.82	501.36	496.82	168.32	990
17	727.27	181.82	502.27	496.82	178.22	990
18	818.18	181.82	503.18	496.82	188.12	990
19	909.09	181.82	504.09	496.82	198.02	990
20	90.91	272.73	495.91	497.73	207.92	990
21	181.82	272.73	496.82	497.73	217.82	990
22	272.73	272.73	497.73	497.73	227.72	990
23	363.64	272.73	498.64	497.73	237.62	990
24	454.55	272.73	499.55	497.73	247.53	990
25	545.46	272.73	500.46	497.73	257.43	990
26	636.36	272.73	501.36	497.73	267.33	990
27	727.27	272.73	502.27	497.73	277.23	990
28	818.18	272.73	503.18	497.73	287.13	990
29	909.09	272.73	504.09	497.73	297.03	990
30	90.91	363.64	495.91	498.64	306.93	990
31	181.82	363.64	496.82	498.64	316.83	990
32	272.73	363.64	497.73	498.64	326.73	990
33	363.64	363.64	498.64	498.64	336.63	990
34	454.55	363.64	499.55	498.64	346.54	990
35	545.46	363.64	500.46	498.64	356.44	990
36	636.36	363.64	501.36	498.64	366.34	990
37	727.27	363.64	502.27	498.64	376.24	990
38	818.18	363.64	503.18	498.64	386.14	990
39	909.09	363.64	504.09	498.64	396.04	990
40	90.91	454.55	495.91	499.55	405.94	990
41	181.82	454.55	496.82	499.55	415.84	990
42	272.73	454.55	497.73	499.55	425.74	990
43	363.64	454.55	498.64	499.55	435.64	990
44	454.55	454.55	499.55	499.55	445.55	990
45	545.46	454.55	500.46	499.55	455.45	990
46	636.36	454.55	501.36	499.55	465.35	990
47	727.27	454.55	502.27	499.55	475.25	990
48	818.18	454.55	503.18	499.55	485.15	990
49	909.09	454.55	504.09	499.55	495.05	990
50	90.91	545.46	495.91	500.46	504.95	990
51	181.82	545.46	496.82	500.46	514.85	990
52	272.73	545.46	497.73	500.46	524.75	990
53	363.64	545.46	498.64	500.46	534.65	990
54	454.55	545.46	499.55	500.46	544.55	990
55	545.46	545.46	500.46	500.46	554.46	990
56	636.36	545.46	501.36	500.46	564.36	990
57	727.27	545.46	502.27	500.46	574.26	990
58	818.18	545.46	503.18	500.46	584.16	990
59	909.09	545.46	504.09	500.46	594.06	990
60	90.91	636.36	495.91	501.36	603.96	990
61	181.82	636.36	496.82	501.36	613.86	990
62	272.73	636.36	497.73	501.36	623.76	990
63	363.64	636.36	498.64	501.36	633.66	990
64	454.55	636.36	499.55	501.36	643.56	990
65	545.46	636.36	500.46	501.36	653.47	990
66	636.36	636.36	501.36	501.36	663.37	990
67	727.27	636.36	502.27	501.36	673.27	990
68	818.18	636.36	503.18	501.36	683.17	990
69	909.09	636.36	504.09	501.36	693.07	990
70	90.91	727.27	495.91	502.27	702.97	990
71	181.82	727.27	496.82	502.27	712.87	990
72	272.73	727.27	497.73	502.27	722.77	990
73	363.64	727.27	498.64	502.27	732.67	990
74	454.55	727.27	499.55	502.27	742.57	990
75	545.46	727.27	500.46	502.27	752.48	990
76	636.36	727.27	501.36	502.27	762.38	990
77	727.27	727.27	502.27	502.27	772.28	990
78	818.18	727.27	503.18	502.27	782.18	990
79	909.09	727.27	504.09	502.27	792.08	990
80	90.91	818.18	495.91	503.18	801.98	990
81	181.82	818.18	496.82	503.18	811.88	990
82	272.73	818.18	497.73	503.18	821.78	990
83	363.64	818.18	498.64	503.18	831.68	990
84	454.55	818.18	499.55	503.18	841.58	990
85	545.46	818.18	500.46	503.18	851.49	990
86	636.36	818.18	501.36	503.18	861.39	990
87	727.27	818.18	502.27	503.18	871.29	990
88	818.18	818.18	503.18	503.18	881.19	990
89	909.09	818.18	504.09	503.18	891.09	990
90	90.91	909.09	495.91	504.09	900.99	990
91	181.82	909.09	496.82	504.09	910.89	990
92	272.73	909.09	497.73	504.09	920.79	990
93	363.64	909.09	498.64	504.09	930.69	990
94	454.55	909.09	499.55	504.09	940.59	990
95	545.46	909.09	500.46	504.09	950.50	990
96	636.36	909.09	501.36	504.09	960.40	990
97	727.27	909.09	502.27	504.09	970.30	990
98	818.18	909.09	503.18	504.09	980.20	990
99	909.09	909.09	504.09	504.09	990.10	990

Appendix 2: Large Scale Benchmark Case Definitions

All three large scale benchmark cases use the same gradient field consisting of 100 Gaussian load troughs. The field strength from all the load troughs at any point is calculated in the same way as the standard benchmarks. The field of 100 troughs is constructed by replicating a parameterized group of 10 troughs. The parameters for these 10 troughs are shown in Table 10. Table 10 serves as a 10 source template to replicate in the space. The x_i and y_i values in the table are *relative coordinates* with respect to the position of the first source. The *absolute* position of the first source of each replication is shown in Table 11. The coordinates in Table 11 provide the locations to replicate each 10 source template.

Table 10. Large Scale Source Parameters

Source *i*	x_i	y_i	I_i	σ_i
1	0	0	25	35
2	0	-300	5	35
3	200	150	10	35
4	200	-150	10	35
5	200	-450	10	35
6	400	150	10	75
7	400	-150	10	75
8	400	-450	10	75
9	600	0	25	75
10	600	-300	5	75

Table 11. Absolute Position of the Source Replications

Replication	x_1	y_1
1	90	600
2	1200	600
3	2310	600
4	660	2100
5	1770	2100
6	660	1200
7	1770	1200
8	90	2700
9	1200	2700
10	2310	2700

Appendix 3: Large Scale Benchmark Case Definitions with Noise

Pseudo-random noise is added to the original large scale benchmark field by applying a discrete *noise mask* throughout the mesh. A grid of 100 x 100 random numbers uniformly distributed in the range [-10, 10] is generated using Microsoft Excel. The table of random numbers is too large to print here but can be accessed online. The noise mask is replicated throughout the mesh by using the grid as a lookup table and adding the corresponding random value to each unit of the original benchmark field. To determine the appropriate table indexes for the random number lookup, modular arithmetic is performed using the x- and y-locations of the processor in the mesh. (x mod 100, y mod 100). For example, the random number that is added to the processor at position (373, 1988) in the mesh is at position (73, 88) in the lookup table.