AFRL-IF-RS-TR-2005-125
Final Technical Report
April 2005

COMPUTATIONAL RESILIENCY

Syracuse University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K447

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-125 has been reviewed and is approved for publication

APPROVED: /s/

JOHN C. FAUST
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	APRIL 2005	Final Jun 00 – Oct 03

4. TITLE AND SUBTITLE
COMPUTATIONAL RESILIENCY

5. FUNDING NUMBERS
C - F30602-00-1-0574
PE - 62301E/63760E
PR - K447
TA - 15
WU - A1

6. AUTHOR(S)
Steve J. Chapin and
Susan B. Older

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Syracuse University
Department of Electrical Engineering and Computer Science
Syracuse New York 13219

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency AFRL/IFGB
3701 North Fairfax Drive 525 Brooks Road
Arlington Virginia 22203-1714 Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2005-125

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: John C. Faust.IFGB/(315) 330-4544/John.Faust@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

The Computation Resiliency project investigated the warfare-hardening of distributed scientific applications of interest to the US military through the research and implementation of a middleware layer supporting multithreading, group communication, replication, fuzzy agreement, functionality splitting and merging, checkpointing and restart, and migration. In order to realize such a resilient computing model, an application-independent, distributed middleware library, the Computational Resiliency Library (CRLib), was developed and implemented. The CRLib provides distributed systems with the ability to sustain operation and dynamically restore the required level of assurance in system function during attacks or failures. CRLib supports decomposition of an application into multiple replicated, communicating threads with the ability to restore mission readiness in the face of partial failure. The base CRLib was successfully integrated with three applications: a towed sonar-array application, heat diffusion, and an image processing application. The original goals of the project also included process camouflage and integrated scheduling support to automatically respond to failures and intrusions. Because of early termination, these goals were not met.

14. SUBJECT TERMS
Computational Resiliency, Fault Tolerance, Intrusion Tolerance, Distributed Computing, Replication, Multithreading, Information Warfare

15. NUMBER OF PAGES
71

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Appendixes

1 Introduction

The Computational Resiliency project developed mechanisms to allow applications To respond to attacks and faults, thereby restoring application readiness and ensuring continued operation. These mechanisms include support for replication, migration, camouflage, and mutation. The deliverables on this contract include software prototypes, formal models, and the information warfare-hardening of two applications of interest to DARPA/DoD.

The project was terminated after 18 months (of the 42 proposed). Therefore, the goals and milestones achieved reflect the diminished time scale for the project.

During this project, we developed a core library supporting Computational Resilience. This library includes automated support for replication, migration, agreement, and functionality mutation in distributed scientific applications.

As originally specified, the goals and milestones of the project were as follows:

Formal models

- Core calculus
- Resource policy mechanism
- Equivalence notions
- Protocol analysis
- Extensions and analysis

Software Development

- Extend SCPlib Migration
- Simple camouflage and decoys
- Functionality mutation
- Advanced camouflage
- Policy frameworks
- Computational Resiliency-aware schedulers

Integration of two applications of interest to the DoD with the Computational Resilience library

The first year of the project was spent solidifying the base library, adding thread replication, and extending migration. The actual achievements deviated somewhat from the initial goals, as it was necessary to ensure that the base library had sufficiently reliable and secure base so that the higher-level services would have a solid foundation.

The goals achieved for the library included:

- Passive replication with checkpointing for CRLib applications
- Active replication at a user-settable level
- Split/merge for replicated thread groups
- Message authentication using DSS
- Fuzzy agreement protocols
- Synchronous and asynchronous liveness checking

2 Lessons Learned

The primary lesson taken away is our dependence on the difficulty in increasing the resiliency of an application in isolation from the underlying system, and in the face of a determined, patient adversary. Our approach works well for attacks and failures that occur in the context of the application, but we were faced with a dilemma:

- Assume that the base system provided stability and resistance to external attacks, and concentrate on the goals as put forth in the original proposal, thereby ignoring large classes of attacks, and leaving ourselves susceptible to them, or

- Fortify the base system before proceeding with the development of the scheduling and camouflage systems.

In particular, we felt it necessary to deal with the issues involving agreement, splitting and merging functionality of all threads in a group (necessary for future work in camouflage), and additional replication schemes (necessary to support scheduling alternatives in future work) before proceeding.

In its current state, our Computational resilience library will deal with classes of attacks and failures that:

- disable a subset of the threads in a group (either by killing processes or crashing the machines where they run), in a fail-sop sense:
- compromise a subset (up to 1/3 the total threads; 1/2 if the optional DSS authentication is used) leading to Byzantine failures;
- attempt to use man-in-the-middle techniques or spoofed messages to disrupt computation

In either of the first two cases, the number of replicated threads will be restored to the prior level at the next liveness check. The use of signed messages for authentication thwarts the third class, and increases resilience against Byzantine failures.

The CRLib will not help in the face of an adversary who is patient, stealthy, and puissant-such an adversary that might:

1. break into multiple systems (potentially all of them where our jobs are running)

2. map the system and observe which processes are running CRLib jobs

3. coordinate an attack sufficient to compromise more than 1/2 to cause Byzantine failure, or to kill all threads in a group.

It is because of this type of attacker that we specified a "safe zone" in the original proposal. If we can rely on such a zone, then we can limit our exposure to these attacks by ensuring that less than 1/2 of our threads run outside the zone. It is unclear to us whether this is a reasonable stance to take.

The continued development of camouflage techniques may provide additional protection against these attacks. Even with excellent camouflage, there will be valid reasons to run "in the open" under a low threat condition. In that case, it is critical that intrusion detection systems recognize attacks and reconnaissance activity extremely early, so that camouflage can be enabled before the attackers have compromised many systems, giving them the ability to observe system transitions from uncamouflaged to camouflaged execution, thereby rendering the camouflage ineffective.

3 Technical Achievements

The technical achievements of the project are summarized in four appendixes, which are attached.

Appendix A

Analysis of Deception

Ibrahim Taner Okumus, Haizhi Xu,
Steve Chapin, Susan Older,
Cheol-min Hwang, Joohan Lee, Norka Lucena

Syracuse University

What is a deception?

We define deception as those actions executed to deliberately mislead one's adversary. A special case is when one misleads opposing military decision-makers as to friendly military capabilities, intentions, and operations, thereby causing the adversary to take specific actions that will contribute to the accomplishment of the friendly mission. Deception is virtually anything that assists in creating a disadvantageous misperception in the mind of adversary. Deception techniques generally fall into two categories: morphological (the form of a thing) and behavioral (how it acts).

Our adversary possesses certain structures of perception, which he employs in a strategy of perception. Keeping this in mind, here is a simple definition of deception: *Deception occurs when the designs embedded in the morphology and/or behaviors of one entity defeat the designs embedded in the perceptual structures and/or strategies of another entity* [3]. In conventional camouflage, the structure of perception is the human visual system, which uses regular shape recognition and color to differentiate objects from the background cover. Our understanding of that perception mechanism allows us to design morphological camouflage that is ideally suited for the background conditions.

Historical use of deception for information protection

Common deceptive protection techniques are [2]: Concealment, camouflage, false and planted information, ruses, displays, demonstrations, feints, lies and insights.

Concealment:

Concealment implies that there is a natural environment that is expected by an attacker and that whatever is concealed is hidden somehow by that natural environment. An exhaustive search by the adversary can always find the hidden object. Examples of concealment include:

Concealed services are provided to those who know how to access them. Menu items that don't appear in the menu window, processes that don't appear in a process table, or even "cheat codes" in video games are examples of this kind.

With *path diversity*, multiple paths are used to reduce the dependency on a single route of communications or transportation. The actual path can be hidden in a large space of possible paths.
Steganography is the hiding of one's data stream in another set of data. A common technique embeds communication in an image, in such a way that the altered image is indistinguishable from the original, to the naked eye. A less developed area is the practice of protocol steganography, where one embeds a communication stream within an innocuous-seeming traffic exchange.

Retaining confidentiality of security status information: Protecting the information on methods used to protect the system makes successful and undetected attacks more difficult. This includes not revealing the specific weaknesses of the system.

Camouflage:

Camouflage is based on creation of an artificial cover that makes it appear as if one thing is in fact another for the purpose of making it harder to find or identify. One can also employ *noise injection* to modify the background environment. This makes it harder for an attacker to get a clearer picture of what he is after, but can have the adverse effect of drawing the attacker's attention. This effect can be used as a ruse (see below).

False and Planted information:

This is based on the notion that attacks depend for their success on information about their victims, and that by providing inaccurate information to the attacker, the rate of successful and undetected attacks drops dramatically. Determining what fictions are desired would seem to be most effective if they are part of a strategic plan to cause attackers to act differently that they would act if those fictions were not in place. Creating effective fictions involves understanding the intelligence capacities of the attacker and finding ways to cause their intelligence operations to go awry in desired ways. This is another way of saying that understanding the perceptual structures and strategies of our adversary will allow us to better deceive them.

Perception management: False information is planted though the provision of indicators that would tend to lead attackers to incorrect conclusions. Showing that security measures are tight even though they aren't will prevent most attacks. Showing a valuable resource as less valuable is another example.

Rerouting attacks: Attacks are shunted away from the most critical systems. Using honey pots, lightning rods, and jails are examples of this technique.

Ruses:

Ruses are normally used to cause attackers to believe that they are observing friendly forces when in fact they are not. An example of this is the use of *decoys*, which appear to

be friendly forces (or processes) but whose primary purpose is to draw the attention, and perhaps the attacks, of one's adversaries. A secondary purpose of the decoy can be to serve as a rudimentary intrusion detection system.
Another example of a ruse would be to use noise injection in one area to divert the attention of an attacker away from the actual activity, which might be concealed.

Displays:

The objective of displays is to make the attacker see what isn't there, for example advertising a nonexistent intrusion detection system in entry banners. A disadvantage of displays is that it is hard to make effective displays against sophisticated attackers because they have a tendency to do more intelligence over a longer period of time than amateurs.

Encryption: Encrypting data is a form of display that may discourage the attacker because of the apparent work involved in obtaining the information. If non-critical information is also encrypted, the attacker cannot tell the "wheat" from the "chaff" and may become discouraged.

Lies:

The objective of a lie is to either convince the enemy that something that is not true is true or to convince them they will not get reliable information by asking for it. If the lie is important, it has to be backed up with something, and this is where the false and planted information comes in.

Feeding False Information: False information is fed to the attacker to reveal the presence and identity of the attacker. A well-known example of this occurred in the case documented by Cliff Stoll in *The Cuckoo's Egg*. In that case, to track a hacker who was searching computers for SDI information, Dr. Stoll created false documents and persona. The hacker copied these files, and a few weeks later, a crony of the hacker wrote to the nonexistent person asking for copies of imaginary documents.

Traps: Traps are devices to capture the attackers interest and keep the attacker busy while additional information about the attacker is obtained by observing the activities of the attacker. This method helps to identify the attacker if the attacker has a certain pattern and keeps the attacker busy somewhere else in the system while additional countermeasures are deployed.

Insight:

Insight allows us to deceive an opponent by out-thinking him. In order to do this, the defender should get as much information about the attacker as possible. Insight involves the psychological ability to understand what deception will work.

ATTACKERS

We note several types of attackers:

Joy riders: Bored people looking for some amusement that break into systems just for the fun of it. They are not malicious, in general. They usually don't have many resources.

Vandals: People that break into systems with the specific purpose of causing some damage.

Scorekeepers: People that break into something well known, well defended, or especially cool, in order to get more points.

Professionals: While the prior three classes can be classified as amateurs, professionals work for an organization or a government to get classified information or to damage critical applications. Professionals have sophisticated and powerful computational resources.

Attacker's motivation:

Curiosity is one of the biggest motivations amongst joy riders and scorekeepers. These types of attackers just want to hack into a system out of curiosity. Trying new hacking tools and hacking into specific operating systems are examples.

Ego/glory is another important motivation of attackers. Usually vandals and scorekeepers attack systems to prove to themselves that they can do it and satisfy their egos. Attacking a well-known and well-protected system, such as that belonging to a computer security expert, is a big ego satisfier.

Malice is another motivation. Vandals, as the name implies, choose destroying or damaging the system as their goals. Destroying data, hardware, and denial of service can be examples of motivation for this kind of attack.

Information theft is one of the serious motivations. In most of the cases the attacker has financial reasons to steal information. Stealing technology, stealing credit card information, stealing company data are examples. Spying can be another motivation. Professionals working for governments or companies attack the systems to get classified/secret information.

Invasion is one of the motivations for professional attackers. Invading a system has lots of benefits. One of them is to have a (resourceful) system to launch attacks to another system. This makes it harder to track the attacker.

What are the sensors of attackers?

The attackers can use built-in utility programs, network scanners, packet sniffers, remote controlled Trojan horses, denial-of-service tools, password crackers, and security probes and exploits [7]. The built-in utility programs that the attackers can use are: ps, top (list the running processes), gdb, vmstat (show virtual memory usage), whois, ping, finger (list information about a host), netstat (display network connetctions), showmount (identify mounted resources), telnet, tftp in UNIX; and nbstat (list user accounts on Windows host), net user (list user accounts on Windows domain), net view (list computer belonging to Windows domain and shared resources on the hosts).

The network scanners that the attackers can use are: nmap (port scanner), SATAN, ncat, mscan, Saint, etc. These tools are used to scan for the well-known vulnerabilities in programs such as sunrpc, mountd, netbios, pop3, etc. The packet sniffers that the attackers can use are: Etherfind, tcpdump, esniff, NetXRay, Network Monitor (in windows systems) [9]. There are also some remote-control-Trojan-horses for the attackers to use. These tools include BackOrifice and NetBus [10]. There are also many denial-of-service tools available to the attackers. These tools are TCP Syn, smurf, teardrop, ping-of-death (work on windows 95), Boink, etc. The attacker can also use password crackers such as netcrack and crack (for UNIX), L0phtCrack, and ScanNT for Windows.

In addition, the attackers can make use of the existing exploits to probe the system. Such exploits include Microsoft IIS server 4.0 buffer overflow attacks, sendmail overflow, etc. There are so many existing exploits and attacking codes on line. The attacks are very easy to launch using existing exploit tools. Note: the attackers may have more attacking tools than the list above. The above list is a comprehensive but not complete list of the hacker's tools.

The attacker could have root privileges. Once the attacker has root privileges it is trivially easy to discover detailed information about the system (in our case whether our process is running in that system and what communication paths we have etc.). This is natural, because one of the primary jobs of a system administrator is to monitor the system. In this way, administrator-level access can be thought of as the ultimate sensor for a local system.

We are concerned with attackers who, either as a primary goal or as a secondary effect of their attack, will find our critical computation and stop it. To achieve this goal the attacker can first observe the environment we are running on. The attacker can analyze the traffic and analyze the computers to profile the system and get information as to whether there is anything worthy of attack or not. Once the attacker identifies that the environment is one where our computation might be (or likely is), then his task becomes one of finding our computations and stopping them. Based on this scenario, our goal in deploying camouflaging techniques is to hide from observation, thereby short-circuiting the path of our adversary.

To rate the potential value of camouflage, we must ask what we want to protect or camouflage. In our case, we are protecting a distributed computation, comprising

multiple processes and their communications in a distributed environment (ranging from a cluster to the Internet).

Based on the above analysis, we consider that any or all of the following may be true:

1. The attackers can view all traffic between processes (e.g. they are employing a network sniffer). They can statistically monitor the traffic and specifically check a few related packets.
2. The attackers have root permission on some, but not all, of the computers in the system. Thus, they have incomplete knowledge of our application.
3. The attackers are employing traffic analysis in an attempt to find out the source and the destination of the communication and the content that is being transferred.
4. The attackers have limited analysis tools or time and ability to check large volume network traffic with unspecified patterns. This may be because they wish to avoid having their activities detected, or because the information they are scanning for has a limited useful lifetime (we can enhance our odds of the latter by reconfiguring our application periodically).

Depending on the environment our computation is running in, we can deploy different levels of camouflage techniques. Based on the above assumptions, we can make our processes have patterns similar to the existent commonly used software and traffic.

As an example, we obtained some statistics from the Internet in order to understand the traffic patterns in a regular network:

- For a .edu website [4], the HTTP traffic per day is 302 Mbytes, with HTTPS traffic of 1.724 Mbytes, which is 0.3% of the total http traffic.
- For a commercial website [5], the HTTP traffic per day is 1595 Mbytes, with HTTPS traffic of 422.373M, or 27% of the total http traffic.

On a transpacific link, traffic averages [6]:

TCP: ~87% of Packets (but ~95% of data)
HTTP: ~90% of the TCP data (85% of total data)
Secure HTTP: ~10-15% of Packets (but ~40% of data)
Conventional HTTP: ~85-90% of packets (but ~60% of data)

This data is especially interesting because it indicates that HTTP traffic dominates the net (almost 80% of total traffic), and that large, secure HTTP packets are quite common (and in fact account for about 1/3 of the total data passed on the net).

Which deception techniques work for which kind of attacks/attackers?

The following table summarizes our judgment of the effectiveness of various types of deception against different classes of attackers. Each entry is on a numerical scale of 1

(trivial to defeat) to 5 (nearly impossible to defeat); in other words, higher numbers indicate that that defense mechanism should work well against that group of attackers. In some cases, we have a range of values because of the differing kinds of mechanisms included in that category.

	Joy riders	Vandals	Scorekeepers	Professionals
Concealment	5	4	4	2-4
Camouflage	5	4	3	2
False info	5	4	3	3
Ruses	5	4	4	2
Displays	5	5	4	3
Lies	5	5	4	3
Insight	5	4	4	4

Most of the deception techniques will work against amateur attackers. Our major concern is to protect our computation from professionals. Professionals work patiently, slowly and carefully before they attack a system. Considering the resources an attacker might have we can implement different types of deception techniques.

Concealment is an effective technique against non-professional attackers. As we mentioned an exhaustive search can always find the hidden object, but only professional attackers will have the time and resources to do this. Camouflaging techniques have same effect on attacker types. Unless an attacker observes the system very carefully, it is hard to recognize the camouflaged element. Some camouflaging techniques may attract attention to our system (for example, noise injection) but still make the detection of the actual object difficult. The planting false information technique has a different effect on attackers. Putting a decoy or a honey pot into the system to attract the attention of an attacker is an example application of false and planted information. In the case of using honey pots and lightning rods, a non-professional attacker possibly easily falls for the trap while a professional attacker might get suspicious. Using these sorts of attraction sources will help divert the attention of a non-professional attacker and avoid those sorts of attackers while telling the professional attacker that there is something worthy of protection around there, but it is also being protected and the system is being monitored. Thus either will avoid or delay professional attacks. Ruses and displays are also effective against non-professional attackers.

It is very hard to defend a system against a well motivated, sophisticated, and computationally powerful attacker. Our goal is to make the computation as less recognizable and findable as possible while both diverting and scaring most of the attacks or gaining enough time to remap and/or relocate our computation in the presence of an attacker. With the help of an intrusion detection system we believe that these methods will have considerable impact on the survivability of our computation.

References

1. Y. Guan, et, al. NetCamo: Camouflaging Network Traffic for Qos-Guaranteed Mission Critical Applications. Texas A&M.
2. Cohen, Fred; A note on the Role of Deception in Information Protection. http://www.all.net/journal/ntb/deception.html
3. http://www.rand.org/natsec_area/products/animal.html
4. http://www.uakron.edu/usagestats/
5. http://www.ripe.net/ripencc/pub-services/stats/site/
6. http://tracer.csl.sony.co.jp/mawi/
7. http://net-services.ufl.edu/~security/hacker.html
8. http://www.dataguard.no/bugtraq/1994_3/0232.html
9. http://packetstormsecurity.org/sniffers/
10. http://www.netbus.org/main.html
11. http://www.pbs.org/wgbh/pages/frontline/shows/hackers/whoare/tools.html
12. http://www.engarde.com/~mcn/intrusion/tools/
13. http://www.insecure.org/nmap/
14. http://www.cultdeadcow.com/tools/
15. http://www.nmrc.org/faqs/hackfaq/hackfaq-5.html
16. http://www.securityfocus.com/tools/7
17. http://www.nmrc.org/faqs/hackfaq/hackfaq-8.html

Appendix B

Reliable Heterogeneous Applications

Joohan Lee, *Member, IEEE*, Steve J. Chapin, *Member, IEEE*, and Stephen Taylor

***Abstract*—This paper explores the notion of *computational resiliency* to provide reliability in heterogeneous distributed applications. This notion provides both software fault-tolerance and the ability to tolerate information-warfare attacks. This technology seeks to strengthen a military mission, rather than to protect its network infrastructure using static defense measures such as network security, intrusion sensors, and firewalls. Even if a failure or attack is successful and never detected, it should be possible to continue information operations and achieve mission objectives. Computational resiliency involves the dynamic use of replicated software structures, guided by mission policy, to achieve reliable operation. However, it goes further to regenerate, automatically, replication in response to a failure or attack, allowing the level of system reliability to be restored and maintained. This paper examines a prototype concurrent programming technology to support computational resiliency in a heterogeneous distributed computing environment. The performance of the technology is explored through two example applications.**

***Index Terms*—Computational resiliency, distributed system, fault tolerance, information warfare, load balancing, network security.**

ACRONYMS[1]

AFRL	Air Force Research Laboratory
BT	Base-T
DARPA	Defense Advanced Research Projects Agency
HeteroG	heterogeneous
HomoG	homogeneous
HYDICE	hyper-spectral digital imagery collection experiment
IW	information warfare
LAN	local area network
LB	load balancing
LINUX	a kind of free UNIX-type operating system
MPI	message passing interface
No-LB	no LB
Pentium	a kind of microprocessor by Intel for a personal computer
PVM	parallel virtual machine
SCPlib	scalable concurrent programming library
s-PCT	screening principal component transform
UNIX	a kind of operating system for computers such as SUN work-stations
Windows NT	a kind of operating system developed by Microsoft

NOTATION

C_i	capacity of computer i
i	identification of a computer
j	identification of a task
l_j	load of task j
L_i	load on processor i
r_j	number of replicas for task j
S_j	set of computers to which task j may not be allocated
T_i	set of tasks mapped to computer i
U_i	use of a computer i

I. INTRODUCTION

ANY system that operates in highly adverse environments, such as battlefield command and control, must be able to operate reliably by tolerating failures and attacks. Many distributed systems have sought to use replication, either in hardware or software, as a mechanism to provide fault-tolerance and recovery. These approaches provide graceful degradation of performance to the point where no further replication is available and then system failure occurs. This is not sufficient to *assure* information operations in adverse military situations where networked resources can become available dynamically through retasking.

An alternative model of distributed computation is being investigated, termed *computational resiliency* [33], [34], [35]. This model combines real-time attack assessment with process reconfiguration, dispersion, and on-the-fly replication to maintain information operations reliably. To visualize how these concepts might operate, consider a distributed application as analogous to an apartment complex inhabited by a new strain of roach (process/thread)[2] . The roaches are highly resilient: you can stomp on them, spray them, strike them with a broom, but you never kill them all or prevent them from their goal of finding food (resources). To foil your eradication efforts, they use several techniques:

- They are *highly mobile*, moving from one place in the apartment complex (network) to another with speed and agility.
- They continually *replicate* to ensure that it is not possible to kill them all.

Manuscript received June 27, 2001. This research was sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under contract N66001-99-1-8922 and the U.S. Air Force Research Laboratory (AFRL), Information Warfare Directorate, Rome Laboratory, NY USA. Responsible Editor: N. Ye.

J. Lee is with the School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816 USA (e-mail: JLee@cs.ucf.edu).

S. J. Chapin is with the Center for Systems Assurance, Department of Electrical Engineering and Computer Science, Syracuse Univeristy, Syracuse, NY 13244 USA (e-mail: Chapin@ecs.syr.edu).

S. Taylor is with the Thayer School of Engineering, Dartmouth College, Hanover, NH 03755 USA (e-mail: Stephen.Taylor@dartmouth.edu).

Digital Object Identifier 10.1109/TR.2003.819502

[1]The singular and plural of an acronym are always spelled the same.

[2]Thanks to Cathy McCullum for providing this analogy.

- They *sense* their environment (attack assessment) to obtain clues that mobility is necessary: if a light is turned on, they scurry away in all directions to hide behind cupboards in places of *known safety* (secure network zones).
- If a new roach killer is invented they *learn* from it, and *adapt* their behavior to compensate. However, this new strain is particularly aggressive and seeks to live in the daylight (wide-area operation): thus it adopts techniques for camouflage as a form of protection and disinformation.

To support this model, an application independent, programming technology that operates in heteroG distributed computing environments, has been developed. The technology can be applied either to an entire application or a few selected components that are crucial to reliable operation. It incorporates the notion of resiliency into an application through a novel message-passing library. The library hides the details of the communication protocols required to achieve automatic on-the-fly replication and reconfiguration. It operates on a broad variety of networked architectures that include commercial-of-the-shelf computer systems and networking components, shared-memory multiprocessors and clusters of homoG machines. The library distinguishes these architectural differences for the purpose of performance improvement: For example,

- when communicating within shared memory, pointer copying is used;
- when communicating within a homoG cluster, no byte or machine translations are needed.

Because machines in the environment can have widely different performance and memory characteristics, LB techniques are required. These techniques must disperse replicated structures to realize improved reliability. To explore the performance issues associated with these concepts, the technology is incorporated into two prototype distributed applications:

- a towed array sonar, and
- a hyper-spectral remote sensor.

This paper outlines the applications, and shows how resiliency is applied to them. Performance measurements are provided that quantify the overhead of resiliency, under usual operating conditions, using a network architecture containing 21 heteroG computers connected with both Gigabit and Fast Ethernet technologies.

II. Related Work

Fault-tolerance and recovery techniques can be implemented in hardware, software, or a combination of both. The concern here is primarily with software based techniques that can be applied to distributed real-time applications, such as battlefield command and control. Most of the techniques developed to date are based on the notion of *process replication* to provide high levels of system availability [1]. Unfortunately, the use of replication introduces additional problems such as:

- the need to maintain consistency between replicas,
- detect the failure of a compromised process,
- transparently recover system function.

In many client-server style applications, the techniques used to provide recovery can be divided into 2 general categories based on *passive* [2] or *active* [3] replication.

- In passive replication [2], there is a single primary source and all other replicas are maintained purely as backups. Only the primary source receives requests from clients and guarantees the ordering and atomicity of message delivery. Although easy to implement, this method is slow to transfer control to a backup in the event of failure; this can lead to appreciable degradation in system response.
- In active replication [3], all replicas have the same level of control. Any viable replica can receive a message from a client and collectively the replicas maintain message ordering and atomicity. This approach is attractive for real-time systems because it provides a more transparent view of the system to client processes and is relatively fast to transfer control in the event of failure [4].

To implement replication it is useful to organize processes into *groups* and provide communication mechanisms between groups. The concept of a process group was first introduced in the V-kernel to express one-to-many communication structures [5]. A group is a set of processes sharing common application semantics, as well as the same group identifier and multicast address. Each group is viewed as a single logical entity hiding its internal structure from other groups. The processes in a group cooperate to provide a single service. In order to maintain and share a consistent process state, the processes use multicast communication primitives that guarantee every process in the group receives the same messages in the same order. The group concept has been extended to many fault-tolerant distributed systems such as Isis [6], Horus [7], Transis [8], Totem [9], and Ameoba [10]. These systems all allow members of a group to fail, thereby providing graceful degradation of performance to the point of system failure. Although not used for fault-tolerance, the process group has also been used widely as a concurrent programming paradigm through libraries such as PVM [11] and MPI [12].

A useful taxonomy of database recovery techniques for information warfare has been developed by Jajodia [13].

- Cold-start recovery involves a complete restart in the event of a severe attack.
- Warm-start involves nontransparent but automated recovery.
- Hot-start techniques are by far the more sophisticated and provide transparent recovery.

These techniques operate through a combination of implementation techniques that include checkpoints and intelligent analysis of the effects of attack queries [13]. Checkpointing generally requires more time to recover than process groups since it involves restoring previous state information from a stable repository such as hard disk and starting a new process. Checkpointing mechanisms can sometimes be used transparently, and a variety of techniques have been developed to reduce the associated overheads [14]–[17].

The use of networks of personal computers, work-stations, and symmetric multiprocessors as a +computing platform requires LB techniques. Computers in a typical network often differ in processor performance and memory characteristics. Many LB techniques have been developed for heteroG environments [18], [19]. These typically assume that attacks

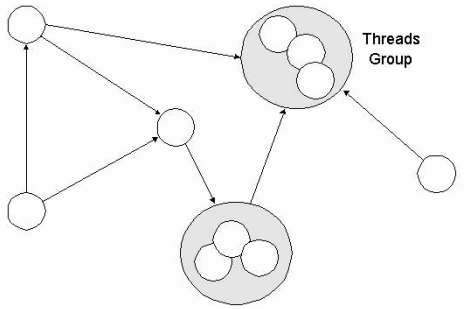


Fig. 1. Replication of threads.

or faults are unlikely, and focus on the optimal allocation of resources. LB techniques for efficient allocation of the replicated processes have also been studied in fault-tolerant systems [20]–[23]. For example, [20] proposes a static model to derive the mapping of replicated processes. [22] presents an algorithm that balances the load of replicated processes over a homoG system and subsequently analyzes the performance of the algorithm. [23] proposes a model that expresses the reliability of the system in terms of the probability that the system can run an entire task successfully. This model introduces a process allocation algorithm that maximizes the reliability over heteroG systems.

The starting point for the work described in this paper is the SCPlib [24]–[26]. This library provides a heteroG concurrent programming technology and has been applied to a variety of irregular, large-scale, industrial simulations [27]. The library is portable to a wide range of platforms, from distributed-memory multicomputers to networks of work-stations, PCs and multiprocessors. It provides a *mobile thread* abstraction in which threads may move between processors to accommodate changes in resource requirements (*e.g.*, processor speed, memory, bandwidth). The communication structure of an application is represented explicitly and can thus be changed transparently as a thread migrates. The library includes a variety of LB and granularity control techniques based on thread migration.

III. Computational Resiliency

To provide reliable operation, applications can choose to replicate selected mission critical threads, thereby forming *thread groups*, as shown in Fig. 1. Each thread in a group is allocated to a different computational resource to sustain operation. This provides a graceful path of performance degradation to the point of failure. Unfortunately, it is not resilient because it does not *assure* continued operation of the system when resources become available dynamically elsewhere in the network. In any realistic system, there will never be sufficient resources to replicate all threads, therefore policy-based methods for controlling replication are required.

An alternative approach is to *automatically recreate the level of thread replication* in the face of failure or attack. This assures that operational reliability is eventually restored, subject only to the constraints imposed by the time-dependent availability of resources. Obviously, to be successful, the replacement thread must be dynamically mapped to an alternative location in the network with sufficient resources. Protocols are required to dynamically-reconfigure communication between residual thread groups and newly created replicas. These protocols deal with race conditions inherent in the reconfiguration process, ensure that no communication is lost, that the integrity of state is maintained, and that where possible locality of communication is preserved.

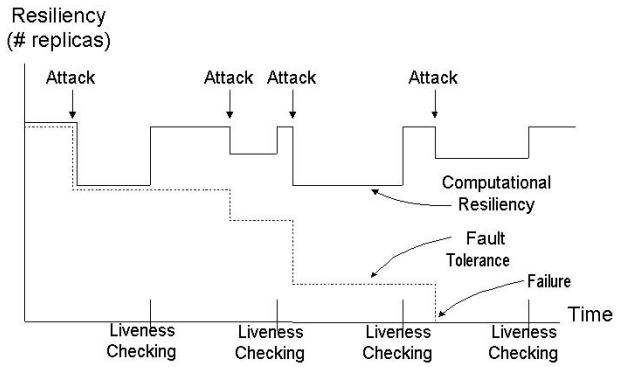


Fig. 2. Fault-tolerance versus computational resiliency.

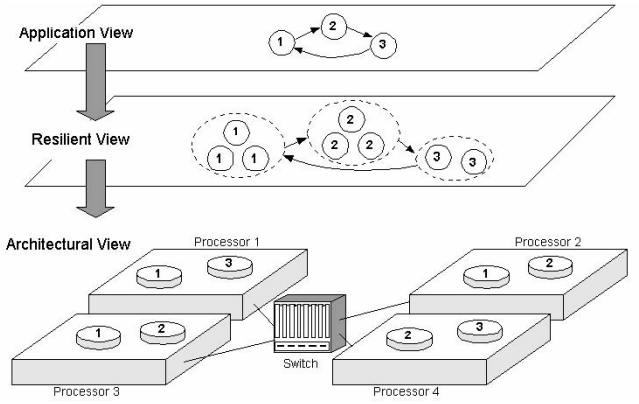


Fig. 3. Computational resiliency using a cluster of multiprocessors.

Fig 2 compares the fault-tolerant model of computation with computational resiliency. In a fault-tolerant implementation (dashed line), as threads fail, graceful degradation occurs and, eventually, when no replicas are available, the application is unable to proceed.

Using resiliency, periodic liveness checks are performed. These checks determine if an application is not performing as anticipated. If an application thread is detected as compromised during a liveness check, it will be destroyed and replaced using the uncompromised residual members of the group. This hot-start recovery mechanism [13] ensures that the newly recreated thread begins execution from the most recent state rather than a state where the compromise occurred. No message logging or intermediate state is saved either in stable storage such as a hard disk, or at a remote server. Therefore, network file system-failure does not affect robustness.

Fig. 3 shows how resiliency is layered into a distributed application. The application programmer simply describes the required thread structure and states the level of resiliency for each crucial thread. In the diagram there are 3 threads, the first and second are resilient to level 3, while the third is resilient to level 2. Communication between threads at the application level is replaced by group communication at the resilient level. Threads are subsequently mapped to appropriate processors such that

```
for all i
    T_i ← φ
end for
for each task j
    S_j ← φ
    while r_j > 0
        choose processor i that is not in S_j and
            has the lowest utilization
        if the reliability constraints are met
            assign the replica of task j to computer i
            L_i = L_i + l_j
            T_i ← T_i ∪ {task j}
            r_j = r_j − 1
        else
            S_j ← S_j ∪ {i}
        end if
    end while
end for
```

Fig. 4. Load balancing algorithm.

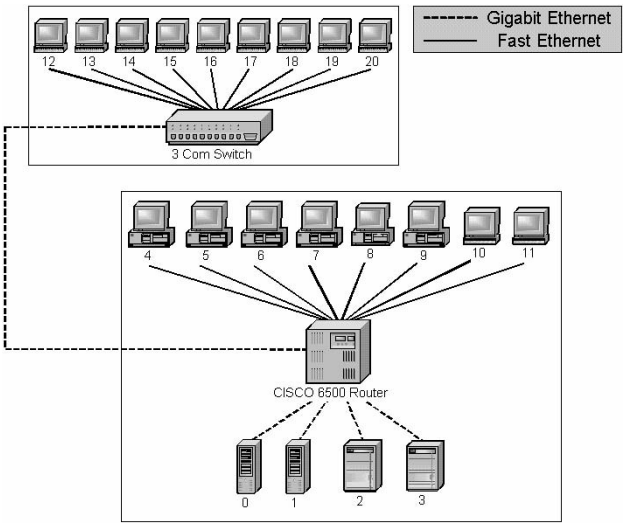


Fig. 5. HeteroG network architecture.

replicas in a single group are placed in different processors at the architectural level.

IV. Load-Balancing Algorithms

Traditional LB techniques address the optimal allocation of resources to tasks. This process is then augmented with reliability constraints. Fig. 4 outlines the greedy algorithm used;

l_j load of task j,
L_i load on processor i,
r_j number of replicas for task, j
T_j set of tasks mapped to computer i,
S_j set of computers to which task j can not be allocated.

To assess the load of a task, measure the execution time of a standard benchmark task on the slowest computer in the network, and assess the relative performance of any other computers. Assume that faster processors have a larger capacity, based on relative speeds, and this allows the algorithm to determine the processor with lowest utilization.

The reliability constraints assure that each replicated task is assigned to a distinct computer because the failure of a computer results in loss of all the replicas in it. In addition, if more than one LAN is in use, loss of network connectivity between LANs can reduce reliability. Thus, always ensure that replicas within a group are allocated to different LANs where possible.

V. Experimental Testbed

To explore the feasibility of these concepts, two prototype applications were developed, and then mapped to a network architecture organized as 21 heteroG computers connected with a 100 BT and Gigabit Ethernet switches. These computers included a broad range of performance and memory characteristics, operating systems, and byte orderings, as follows:

- Machine 0: One 4-processor, Pentium III, 450 MHz machine running Windows NT 4.0, with 1.5 Gbytes of memory, Gigabit network. (9.3)
- Machines 1, 2: Two 8-processor Pentium III, 500 MHz machines running Windows NT 4.0, with 4 Gbytes of memory, Gigabit network. (14.4)
- Machine 3: One dual processor, Pentium II, 300 MHz machine running Windows NT 4.0, with 256 Mbytes of memory, Gigabit network. (2.6)
- Machines 4, 5: Two Pentium III, 500 MHz machines running Windows NT 4.0, with 128 Mbytes of memory, 100 BT network. (2.6)
- Machines 6, 7, 8: Three Celeron 533 MHz machines running Windows NT 4.0, with 128 Mbytes of memory, 100 BT network. (2.2)
- Machine 9: One SGI Indigo II, 200 MHz R4400, running IRIX 6.4, with 128 Mbytes of memory, 100 BT network. (1.0)
- Machine 10: One SGI Indigo II, 150 MHz R4400, running IRIX 6.4, with 288 Mbytes of memory, 100 BT network. (1.3)
- Machines 11 20: Ten dual processor, Pentium II, 400 MHz machines running LINUX 2.2.12, with 256 Mbytes of memory, 100 BT network. (3.1)

The performance of each of these machines was measured relative to the slowest machine, the 200 MHz Indigo II, and is shown in ( ). A small benchmark problem, roughly 20% as large as the full target applications, was run on each machine to assess its relative performance. The performance of the machines varied by a factor of almost 14.4×8, and the available memory varies by a factor of 32. Fig. 5 shows the overall networking structure composed of both Gigabit Ethernet and Fast Ethernet networking. Machines were grouped into 2 separate sub-networks that were connected through the Gigabit Ethernet networking.

VI. Concurrent Sonar Processing

Sonar systems detect, locate, and classify underwater targets by acoustic means [28], [29]. One of the most important processes in sonar operations is beam-forming. This process combines the outputs from a number of omni-directional transducer

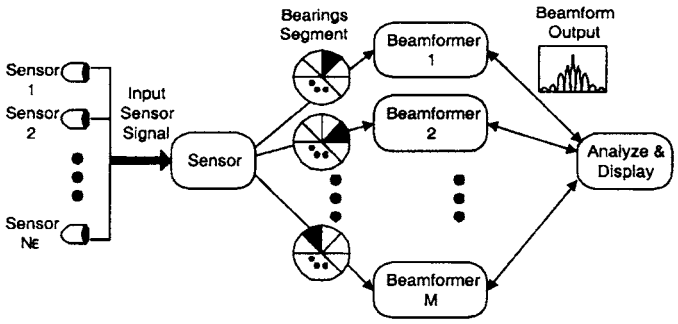


Fig. 6. Communication model for sonar processing.

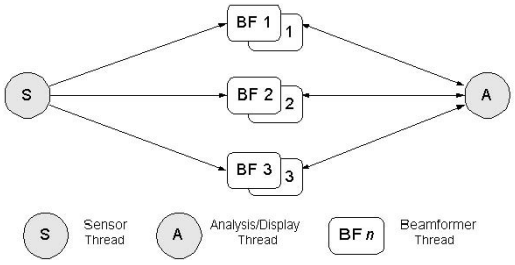


Fig. 7. Resilient view.

elements, arranged in an array of arbitrary geometry, so as to enhance signals from some defined spatial locations. It also suppresses signals from other nontarget obstacles. Beam-formers must be capable of forming and processing large numbers of narrow beams simultaneously to give reasonable angular cover, as well as good angular resolution. In addition, beams must be independently steered and stabilized to compensate for the effect of a ship's motion.

In collaboration with the Ocean, Radar, and Sensor Systems Division at Lockheed Martin, a concurrent towed array sonar application was developed, based on conventional beam-forming techniques [30]. Fig. 6 shows the general concurrent structure of this application. A sensor thread is constructed to simulate the signals emanating from a towed array sonar, containing NE sensor elements. This simulation creates the sonar returns that would emanate from a generic submarine cruising a random path in the Persian Gulf. The 360 degrees of sonar resolution are partitioned among M *beam-former* threads. Each thread fifo-buffers NS partial returns and repeatedly computes a covariance matrix and a partial beam-forming result for the set of angles in the partition. The partial results are combined at a separate thread that performs analysis based on triangulation to determine the track and speed of the target. This thread also presents a waterfall display of the result.

Fig. 7 shows the resilient view of the application where the beam-former threads are replicated with degree 2.

The sensor and display were mapped to Machine 0 in the testbed due to memory concerns, while each of the remaining 20 machines executed beamformers. Resiliency was applied uniformly to harden the application by replicating the beam-forming elements. Figs. 8–10 and Table I show representative experimental results from a broad set of experiments conducted to measure the effectiveness of LB and of the overhead caused by resiliency and liveness checking. The beam-former was executed once for Figs. 8 and 9 and Table I, and 100 iterations for Fig. 10. Each iteration processed a single set of buffered returns. Three parameters were varied in the experiments:

- LB method,
- level of replication (1, 3, or 7),
- frequency of the liveness checking (0 to 20 checks over the course of the 100 iterations).

Even though resiliency 7 might seem to be a high level of replication, this case is interesting to investigate because it more closely approximates the computational model in Section I. The number of sonar elements was fixed to 382, and the number of buffered returns was fixed to 1000.

Three experiments were conducted to evaluate the effectiveness of the various LB techniques.

1) the problem was run No-LB, where equal amount of load is assigned to each processor regardless of their capability.
2) LB strategy based on the number of processors in each machine was used (HomoG-LB).
3) relative performance of each machine measured in Section V was used. Using these static capacity estimates, the problem was then balanced (HeteroG-LB).

Fig. 8 shows the relative use of the computers based on their relative capacity and workload assigned.

U_i use of a computer i,

L_i load of computer i,

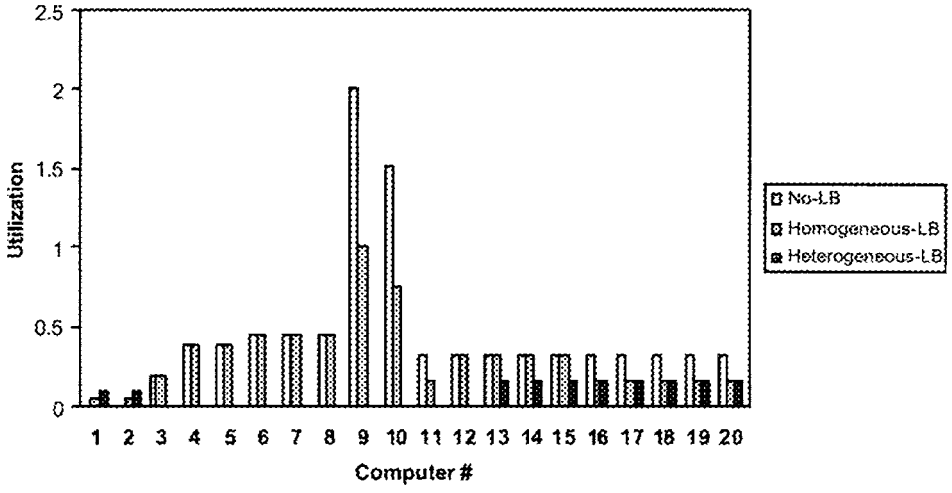
C_i capacity of computer i.

U_i L_i/C_i

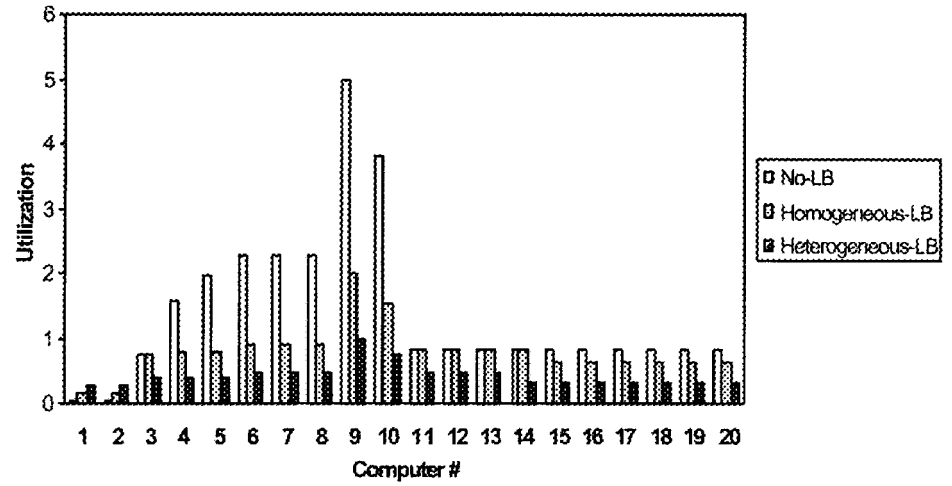
C_i was determined by the relative performance described in Section V; L_i was measured by abstract algorithmic quantities such as the number of operations and size of data structures. For resiliency 1, no task is assigned to a slow processor; in resiliency 3 at least one task is allocated to every computer to ensure higher reliability. For example, in Fig. 8(a), machines 3 to 12 were dropped for resiliency 1. HeteroG-LB technique shows the most balanced use. Machines 1 and 2 have room to take more tasks but cannot take them due to the reliability constraints.

Table I summarizes the results of these experiments. With homoG-LB, the execution time of each beam-forming operation was reduced from 278.9 seconds to 146.9 seconds, which is a 1.9 fold performance improvement with resiliency 7 as in Table I(c). With heteroG-LB, a 2.7 fold performance improvement was observed with resiliency 7 in the same table.

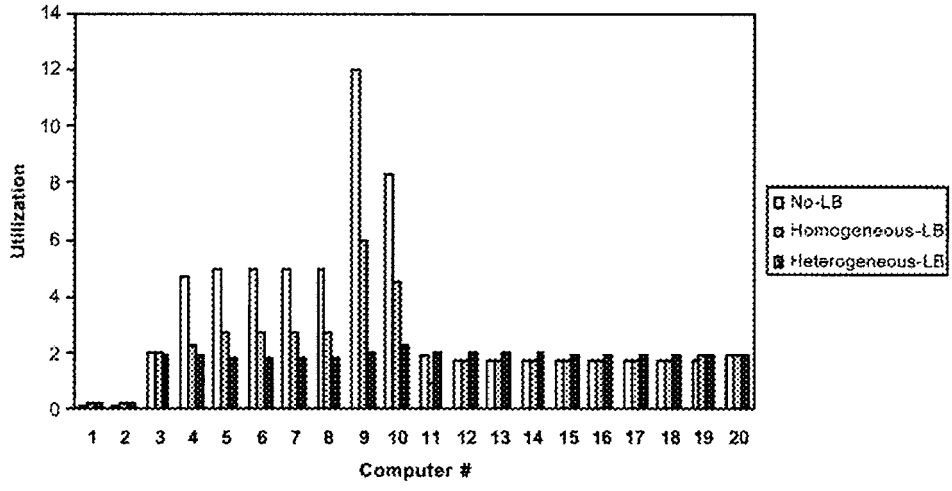
Fig. 9 shows the overhead of resiliency with respect to each LB technique. Execution time in Fig. 9 represents one beam-forming operation. The anticipation was that because replication of a thread doubles its computational requirements, level 3 and level 7 resiliency would execute with a 3-fold and 7-fold decrease in speed, respectively. Without any LB, Fig. 9(b), execution time for resiliency 7 increased more than a factor of 7. With LB, however, the results indicate that performance did not decrease linearly with the level of replication and was less than anticipated for all the cases. The execution time of resiliency 3 increased only 127% and 120% over resiliency 1 for Fig. 9(c) and (d), respectively. For resiliency 7, it was as much as 568% over resiliency 1, indicating that very high levels of survivability



(a)



(b)



(c)

Fig. 8. Use for each resiliency: (a) 1, (b) 3, (c) 7.

might be possible without a direct linear cost. This artifact results from the overlapping of communication and computation in the resilient application: Idle time allows cycles to be used in completing replicated tasks that would have otherwise be wasted. Obviously, this phenomenon is highly application-dependent; however, idle cycles can occur for many reasons in distributed applications, *e.g.*, file I/O, synchronization, and global operations. Therefore it is not unreasonable to assume that resiliency might may often be achievable without important computational costs.

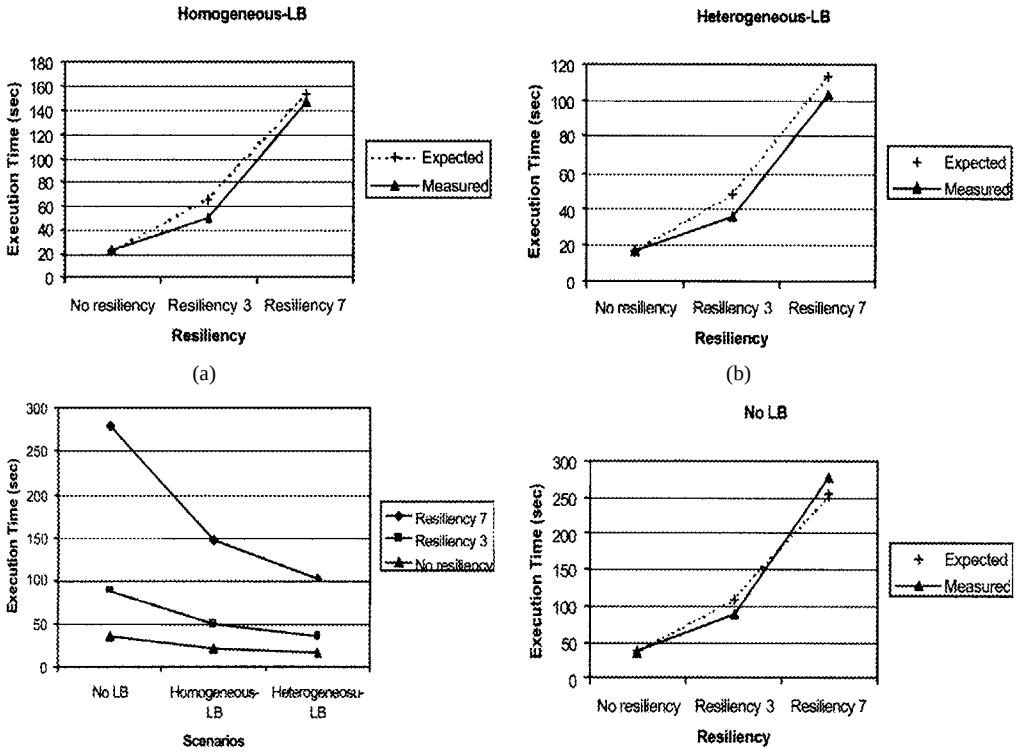


Fig. 9. Overhead of resiliency.

TABLE I
RESULTS OF LOAD BALANCING EXPERIMENTS FOR ENTIRE HETEROG TESTBED

Scenario	Step Time (sec)	Improvement
a. Resiliency 1		
No LB	36.2	N/A
HomoG-LB	22.0	1.65x
HeteroG-LB	16.1	2.25x
b. Resiliency 3		
No LB	88.4	N/A
HomoG-LB	50.0	1.77x
HeteroG-LB	35.4	2.5x
c. Resiliency 7		
No LB	278.9	N/A
HomoG-LB	146.9	1.9x
HeteroG-LB	103	2.71x

Fig. 10 shows the cost of liveness checking in this application. The execution times for 100 iterations of beam-forming operation with respect to the number of liveness checkings were measured. The overheads, ratio of increased execution times with use of liveness checking, never exceeded 1%, even when liveness checking is frequent (once every 5 iterations of the beam-former) and the level of resiliency is high, *viz*, 7.

Also measured was the recovery overhead during the liveness checking in the presence of attacks and failures. Time required to recover from the failure and to recreate a new thread, consists of times to create a new thread at another location, to transfer some system information, and to transfer user specified data structures. The first two elements are common overhead regardless of the applications, while the third element can vary, depending on the applications. The amount of time needed for

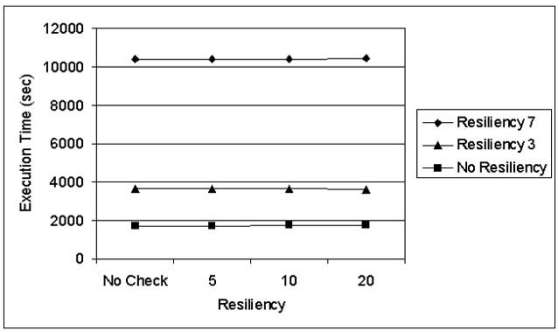
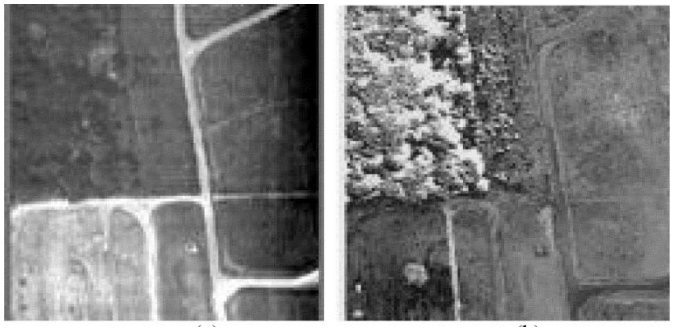


Fig. 10. Overhead of liveness checking.

the first 2 components was measured; it was 3 ms in our experimentation environment.

VII. CONCURRENT REMOTE SENSING

A second application to which resiliency is applied is a concurrent s-PCT that can be used for remote sensing applications [31]. The algorithm takes as input a large number of grey-scale images emanating from a hyper-spectral sensor. Each image corresponds to a particular wavelength of light; *e.g.*, Fig. 11(a) shows the image taken at 1998 nm using a 210-channel hyper-spectral image collected with the HYDICE sensor, an airborne imaging spectrometer. The HYDICE image-set corresponds to foliated scenes taken from an altitude of 2000 to 7500 meters at wavelengths between 400 nm and 2.5 micron. The scenes contain mechanized vehicles sitting in open fields as well as under camouflage. The s-PCT algorithm removes redundancy in the image set and presents a single color composite image that



(a) (b)

Fig. 11. Concurrent remote sensing.

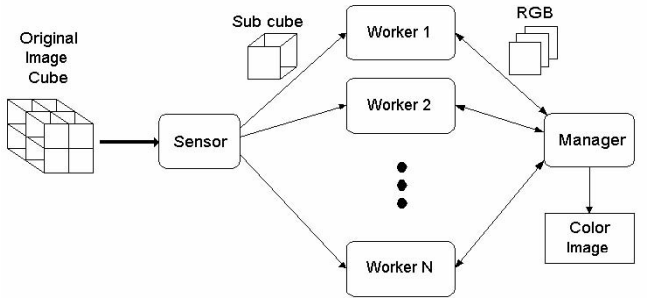


Fig. 12. Manager/worker communication model.

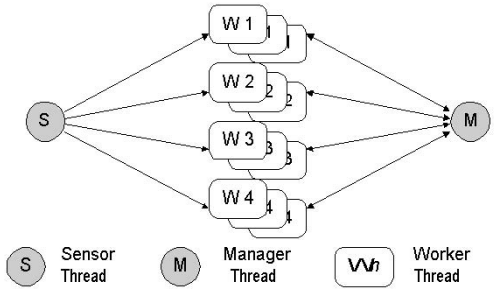


Fig. 13. Resilient view.

shows the important spectral contrast. For example, Fig. 11(b) shows the output of the algorithm in which the mechanized vehicles are clearly visible in the lower left of the figure, due to spectral contrast.

The distributed version of the s-PCT algorithm uses the standard manager/worker decomposition technique [32] as shown in Fig. 12. A sensor thread generates and partitions the 210-frame image cube into sub-cubes, and distributes the sub-cubes to worker threads. A manager synchronizes the actions of these workers, accumulates partial results, and displays the resulting image.

Fig. 13 shows the resilient view of the application where worker threads are replicated with degree of 3.

The performance of the algorithm was measured on the heteroG testbed environment. The same experiment was conducted with all workers replicated up to the level of 7; the manager and sensor were not replicated. Table II shows the results of LB experiments; these are consistent with those in the sonar application. Performance was improved by a factor of 3.37 for resiliency 7. As in concurrent sonar application, higher improvements were achieved with higher resiliency, *viz*, 7.

TABLE II
RESULTS OF LOAD BALANCING EXPERIMENTS FOR ENTIRE HETEROGENEOUS TESTBED

Scenario	Step Time (sec)	Improvement
a. No Resiliency		
No LB	122	N/A
HomoG-LB	81	1.5x
HeteroG-LB	54	2.26x
b. Resiliency 3		
No LB	357	N/A
HomoG-LB	197	1.81x
HeteroG-LB	158	2.26x
c. Resiliency 7		
No LB	896	N/A
HomoG-LB	492	1.82x
HeteroG-LB	266	3.37x

Once again, when resiliency was applied, the anticipated result was that performance would decrease by a factor of 3 or 7, depending on the specified resiliency, because the replicated processes require both memory and processor resources. Fig. 14 shows the overhead of resiliency with respect to 3 LB techniques (no LB case, and 2 LB techniques):

- Without any LB, the execution times for resiliency 7 increased more than a factor of 7 in Fig. 14(b). HeteroG LB reduced the overhead of resiliency appreciably.
- With resiliency 7, the overhead was only 393% over resiliency 1 in Fig. 14(d). As in the sonar application, we observe that LB improved the performance, and resiliency can to use idle cycles in the concurrent algorithm to reduce the cost of replication.

Fig. 15 examines the overhead caused by liveness checking. In each case, the overhead was less than 1% and is consistent with the results from the sonar application.

VIII. DISCUSSION

This paper describes the notion of computational resiliency and discusses the implementation issues associated with a prototype-programming library that supports the idea. It also shows how the concepts and library can be applied in the context of 2 applications: a towed array sonar and a remote sensing application. The applications were studied to ascertain the overheads associated with the technology on a moderately scaled, heteroG architecture consisting of computers with varying computing capability, memory availability, operating systems, and networking technology.

For both applications, use of LB techniques improved the performance by efficient allocation of the replicated threads. Reliability was considered in the LB algorithm to improve the allocation of replicas. The ability to use idle cycles appreciably reduced the cost of increased survivability, especially at higher levels of redundancy than one normally considers. This higher level is directly motivated by the computational model which provides strength in numbers. Although initially, the use of group-based liveness checking was considered to be an important defect with the current implementation strategy, it has proved to be less problematic than anticipated accounting for less than a 1% overhead in both applications.

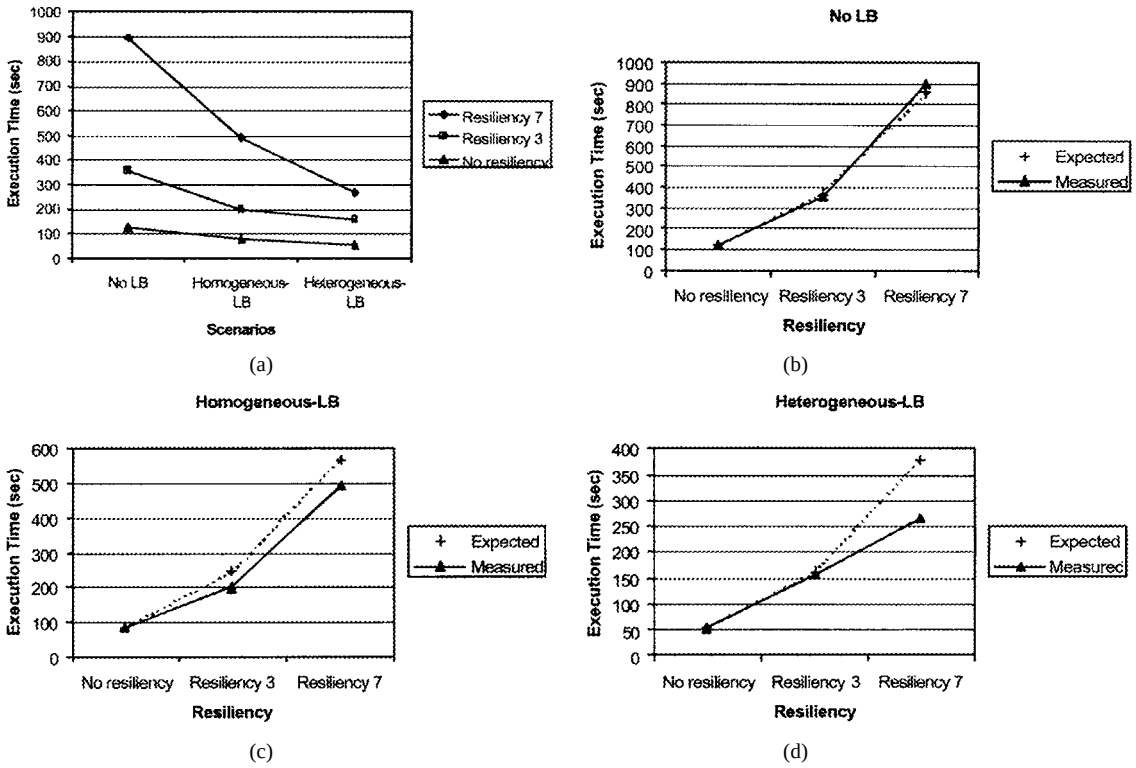


Fig. 14. Overhead of resiliency.

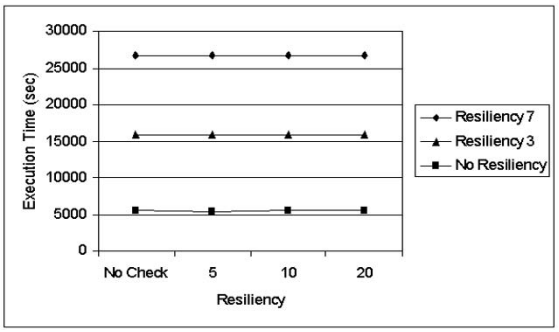


Fig. 15. Overhead of liveness checking.

Many aspects of computational resiliency remain to be explored, and several alternative implementation strategies have yet to be tested. However, this paper indicates that the general concept is both practical and less costly than originally anticipated.

ACKNOWLEDGMENT

The authors thank M. Orlovsky and T. Barnard at the Ocean, Radar, and Sensor Systems Division of Lockheed Martin; G. Irvin and J. Gaska at Mobium Enterprises; and T. Ackalakul and J. Monin at Syracuse University, for their aid in constructing the concurrent applications described in this paper.

REFERENCES

[1] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *IEEE Comput.*, pp. 68–74, Apr. 1997.

[2] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Primary-backup approach," in *Proc. Sixth International Workshop on Distributed Algorithms*, 1992.

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

[4] J. B. Sussman and K. Marzullo, "Comparing primary-backup and state machines for crash failures," in *Proc. Fifteenth Annual ACM Symp. Principles of Distributed Computing*, 1996.

[5] D. R. Cheriton and W. Zwaenpoel, "Distributed process groups in the V-kernel," *ACM Trans. Comput. Syst.*, vol. 3, no. 2, pp. 77–107, Feb. 1985.

[6] K. P. Birman and R. Van Renesse, "Reliable distributed computing with the ISIS toolkit," *IEEE Computer Soc. Press*, 1994.

[7] R. Van Renesse, K. P. Birman, and S. Maffeis, "Horus: A flexible group communication system," *Commun. ACM*, vol. 39, no. 4, Apr. 1996.

[8] Y. D. Amir, S. Kramer, and D. Malki, "Transis: A communication subsystem for high availability," in *Proc. Annual International Symp. Fault-Tolerant Computing*, 1992, pp. 76–84.

[9] D. A. Agarwal, "Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks," Ph.D. dissertation, Dept. Electrical and Computer Eng., Univ. of California, Santa Barbara, CA, 1994.

[10] M. F. Kaashoek, A. S. Tanenbaum, and K. Verstoep, "Group communication in amoeba and its applications," *Distrib. Syst. Eng.*, vol. 1, no. 1, pp. 48–58, Sept. 1993.

[11] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315–339, Dec. 1990.

[12] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming With the Message Passing Interface*: MPI Press, 1995.

[13] S. Jajodia, C. D. McCollum, and P. Ammann, "Trusted recovery," *Commun. ACM*, vol. 42, July 1999.

[14] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Lipckpt: Transparent checkpointing under unix," in *Proc. USENIX Winter Technical Conf.*, 1995.

[15] D. B. Johnson, "Distributed System Fault Tolerance Using Message Logging and Checkpointing," Ph.D. dissertation, Rice Univ., 1989.

[16] B. Ramkumar and V. Strumpen, "Portable checkpointing for heterogeneous architectures," in *27th Int. Symp. Fault-Tolerant Computing*, June 1997, pp. 58–67.

[17] D. J. Scales and M. S. Lam, "Transparent fault tolerance for parallel applications on networks of work-stations," in *Proc. USENIX Technical Conf.*, 1996.

[18] K. Li and J. Dorband, "A task scheduling algorithm for heterogeneous processing," in *Proc. High Performance Computing*, 1997, pp. 183–188.

[19] J. Watts, M. Rieffel, and S. Taylor, "Dynamic management of heterogeneous resources," *High Performance Computing: Grand Challenges in Computer Simulation*, pp. 151–156, Apr. 1998.
[20] L. J. M. Nieuwenhuis, "Static allocation of process replicas in fault-tolerant computing systems," in *Proc. FTCS-20*, 1990, pp. 298–306.
[21] J. Kim, H. Lee, and S. Lee, "Replicated process allocation for load distribution in fault-tolerant multicomputers," *IEEE Trans. Comput.*, vol. 46, no. 4, pp. 499–505, Apr. 1997.
[22] J. A. Bannister and K. S. Trivedi, "Task allocation in fault-tolerant distributed systems," *Acta Informatica*, vol. 20, pp. 261–281, 1983.
[23] S. M. Shatz, J. P. Wang, and M. Goto, "Task allocation for maximizing reliability of distributed systems," *IEEE Trans. Comput.*, vol. 41, no. 9, Sept. 1992.
[24] S. Taylor, J. Watts, M. Rieffel, and M. Palmer, "The concurrent graph basic technology for irregular problems," *IEEE Parallel Distrib. Technol.*, vol. 4, no. 2, pp. 15–25, 1996.
[25] J. Watts, S. Taylor, and S. Nilpanich, "SCPLIB: A concurrent programming library for programming heterogeneous networks of computers," in *IEEE Information Technology Conf.*, 1998, pp. 153–156.
[26] J. Watts and S. Taylor, "A practical approach to dynamic load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, pp. 235–248, 1998.
[27] *Industrial Strength Parallel Computing*: Morgan Kaufmann, 2000, pp. 147–168.
[28] R. Nielsen, *Sonar Signal Processing*: Artech House, Inc, 2001.
[29] T. E. Curtis and R. J. Ward, "Digital beam forming for sonar systems," *IEE (Inst. Electrical Engineers) Proc.*, vol. 127, no. 4, Aug. 1980.
[30] B. Thomas, *Radar and Sonar Signal Processing*, 1998.
[31] T. Achalakul, P. D. Haaland, and S. Taylor, "Mathweb: A concurrent image analysis tool suite for multi-spectral data fusion," in *Sensor Fusion: Architectures, Algorithms, and Applications III*, vol. 3719, SPIE, 1999, pp. 351–358.
[32] L. M. Chandy and S. Taylor, *An Introduction to Parallel Programming*. Boston: Jones and Bartlett, 1992.
[33] J. Lee and S. Taylor, "Advances in computational resiliency," in *IEEE Aerospace Conf.*, Big Sky, Montana, 2001.
[34] J. Lee, S. Chapin, and S. Taylor, "Computational tesiliency," *J. Qual. Reliab. Eng. Int.*, vol. 18, no. 3, pp. 185–199, 2002.
[35] T. Achalakul, J. Lee, and S. Taylor, "Resilient image fusion," in *IEEE Int. Conf. Parallel Processing (ICPP); Workshop on High Performance Scientific and Engineering Computing with Applications (HPSECA)*, Aug. 2000.

Joohan Lee is an Assistant Professor in the Department of Computer Science at the University of Central Florida. He received the B.Sc and M.Sc in Computer Science from Sogang University, Seoul, Korea, in 1993 and 1995, respectively. He pursued his doctorate study at Syracuse University, earning a Ph.D. in Computer Science in 2001. His research interests include fault tolerant distributed systems, high performance parallel/distributed computing, computer security, and computer networks.

Steve J. Chapin is an Associate Professor in the Department of Electrical Engineering and Computer Science, and is Director of the Center for Systems Assurance at Syracuse University. He received the B.S. in 1985 in both Mathematics and Computer Science from Heidelberg College. He pursued his graduate study at Purdue University, earning a Master of Science in 1988 in Computer Science, and a Ph.D. in 1993 in Computer Science. His research areas are operating systems, distributed systems, and computer and network security.

Stephen Taylor is a Professor at Thayer School of Engineering, Dartmouth College. He works for the Institute of Security Technology. His research interests include cyber-forensics, network security, information warfare, real-time sensor fusion, distributed computing and web technologies, high-performance distributed applications, concurrent programming, multi- and hyper-spectral image analysis, and program design methods.

Appendix C

Proceedings of the 2003 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY June 2003

Assuring Consistency and Increasing Reliability in Group Communication Mechanisms in Computational Resiliency

Norka Lucena, Steve J. Chapin, Joohan Lee

Abstract—
The Computational Resiliency library (CRLib) provides distributed systems with the ability to sustain operation and dynamically restore the level of assurance in system function during attacks or failures. In the presence of arbitrary faults, replicated threads need to agree on the values received in order to achieve consistency, when doing group communication in CRLib. To guarantee data integrity and increase reliability, we have implemented a variant of the Lamport-Shostak-Pease oral message algorithm for the Byzantine Generals problem, which provides fuzzy agreement as well as a reduction of the expected communication overhead. Instead of agreeing on the original messages, which could be extremely large, agreement is performed over the 160-bit hashes of normalized messages computed using SHA-1. Performance measurements of applications using CRLib supporting both fail-stop and arbitrary failure models indicate that a reasonable overhead in execution time is worth paying in cases when Byzantine failures are expected.

I. Introduction

Given the increasing number of attacks that result in denial, exploitation, corruption, or destruction of information as well as any activity involving its acquisition, transmission, storage or transformation, it is critical for information systems to have the ability to defend themselves against information warfare (IW) attacks [1]. Researchers define information warfare defense as a continuous process against IW attacks, whose goal is to keep high availability of the system components at any time, and whose phases correspond to a typical protect-detect-react cycle [2]. Therefore, to assure operation, information systems must be resilient, i.e., have the ability to tolerate, recover from, and react to failure and attack.

In general, systems that need to operate under extremely adverse environments use replication of critical data, services and/or resources to increase availability and provide fault tolerance. Although replication of critical information and resources provides graceful degradation of system performance, it is not sufficient to aggressively recover assured operation [3]. The computational resiliency ([3], [4], [5]) model provides distributed information systems with the ability to sustain operation and dynamically restore the level of assurance in system function under IW attacks. This model ensures the restoration of the operational readiness prior to the attacks, subject only to constraints of resource availability.

To realize such a resilient computing model, the authors developed an application-independent, distributed programming middleware library, the Computational Resiliency Library (CRLib), which provides an application-programming interface (API) for concurrent programming, operating in a broad variety of heterogeneous networked architectures. Users of the CRLib only need to specify their particular requirements for reliability. The library transparently takes care of all details regarding communication protocols to achieve on-the-fly replication and reconfiguration. The library also ensures that no communication is lost, that integrity of the process state is maintained, and that, where possible, locality of communication is preserved.

Replication techniques in CRLib are based on the notion of process replication. In particular, due to the concurrent nature of the applications CRLib serves, process replication occurs at the level of thread replication. Several copies of threads are maintained at multiple locations, usually, different computational resources.

To implement replication properly, CRLib organizes threads in groups, which are logical representations of a collection of replicated physical threads. Each thread group, viewed as a single logical unit, hides its internal structure from other groups.

Group communication is based on ordered multicast operations. Although communication details are hidden to the programmer because of the replication transparency provided by the CRLib, multicasting from and to groups of replicated threads leads to replicated messages. A single copy of a thread can receive more than one value, depending on the resiliency (level of replication). In the presence of arbitrary faults, replicated threads need to agree on the values received to achieve consistency.

This study presents a prototype implementation of a so-

This research is sponsored by the Defense Advance Research Projects Agency (DARPA) under contract N66001-99-1-8922 and the Air Force Research Laboratory (AFRL), Information Warfare Directorate, Rome Laboratory, NY.

N. Lucena, S. J. Chapin: Systems Assurance Institute, Syracuse University, 111 College Place 3-114, Syracuse, NY 13244.

J. Lee: School of Electrical Engineering and Computer Science, University of Central Florida, Computer Science Building 227, Orlando, Florida 32816-2362.

lution that supports arbitrary failures in CRLib as well as performance measures to quantify the overhead of dealing with such complex failure model while increasing reliability of the applications.

II. Related Work

A. Failure Models

The capability and effectiveness of fault tolerance techniques depend directly on the assumed failure model. Determining how processes and communication links may fail provides understanding of the effect of failures [6] and the suitability of the implemented techniques. Failure models have evolved through the years and become hybrid. They report the presence of distinct fault modes.

Azadmanesh and Kieckhafer [7] summarized the evolution of hybrid fault models and presented a more detailed hybrid model of five modes. In order of complexity, the models they reported are the Single-Mode Byzantine Model, the Two-Mode Meyer and Pradhan Hybrid Model, the Three-Mode Thambidurai and Park Hybrid Model, the Four-Mode Omissive/Transmissive Hybrid Model and the Five-Mode Omissive/Transmissive Model.

The base of all these models is the *Single-Mode Byzantine Model*, which considers only faults of unrestricted behavior, i.e. faulty processes can send conflicting values to different nonfaulty processes. Such behavior is called *Byzantine* or *asymmetric*.

Meyer and Pradhan [8] divided the space of all possible faults into two subspaces: *Benign* faults and *malicious* faults. Benign faults are known to all nonfaulty processes or can be easily determined, such as crash faults. Malicious faults, on the contrary, are not evident and comprise all the other faults.

Thambidurai and Park [9] decomposed the failure space even more. As Meyer and Pradhan [8] did, they partitioned faults into two disjoint subsets: *Non-malicious* faults and *malicious* faults. However, they also partitioned the space of malicious faults into another two disjoint subsets: *Symmetric* faults and *asymmetric* faults. The difference between these last two is based on how the nonfaulty processes perceive their behavior. The behavior of symmetric faults is perceived identically by all nonfaulty processes, i.e. all nonfaulty processes receive exactly the same value. The behavior of asymmetric (Byzantine) faults may be perceived differently by different nonfaulty processes, i.e. the message might not be received identically.

Azadmanesh and Kieckhafer [10] extended the Thambidurai and Park model, which applies only to synchronous systems, introducing two new fault models that encompass faults in asynchronous systems. They consider *symmetric* and *asymmetric* faults, but divide them disjointly into *omissive* and *transmissive* faults. *Omissive symmetric* faults occur when a process fails to deliver any value to any recipient, but the failure is not diagnosed as in benign faults. Under *transmissive symmetric* faults, processes can deliver incorrect values to the receiving processes, but by symmetry, they all receive the same erroneous message. *Omissive asymmetric* faults refer to situations when a process sends a single correct value to some processes and no value to other processes. *Transmissive asymmetric* faults or Byzantine comprise delivering different erroneous messages to different receivers.

Finally, the same authors [7] added *benign* faults to their four-mode model producing the five-mode omissive/transmissive model.

In order to pursue the most general solution to the agreement problem in CRLib, we assume only Byzantine or malicious faults. Therefore, a faulty process may fail in different arbitrary ways: crashing, sending spurious messages to other processes, lying, not responding to received messages correctly, and so on. Moreover, nonfaulty processes do not know which processes are faulty and they have no way to suspect it.

B. Agreement Protocols

B.1 Agreement Problems

An agreement problem is present whenever processes need to agree on a value previously proposed by one or more processes as the correct one [6]. Under the assumption of Byzantine failures, the key idea of the problem is that the nonfaulty processes must be able to reach a common agreement, even when faulty processes are present in the system.

In distributed systems, there are three well-known agreement problems: the Byzantine Agreement problem (also called Byzantine Generals problem), the consensus problem, and the interactive consistency problem [11]. In Byzantine Agreement, only one process provides the value to be agreed on, and all nonfaulty processors have to agrec on that value. The process that supplies the initial value is usually distinguished as the commander and the other processes as the lieutenants, according to the Byzantine Generals problem presented by Lamport, Shostak, and Pease [12], [13]. In the consensus problem, every process proposes a single value, its own initial value, and all nonfaulty processes must agree on a single common value. In the interactive consistency problem, every processor also proposes a single value but the goal is to agree on a set of common values, one for each process. Table I presents a summary of the problems based on who initiates the value and the final agreement.

The particular characteristic of these problems is that they are closely related. They can be seen as cases of one another, with the Byzantine agreement problem being the base one. Therefore, solutions for the consensus problems and the interactive consistency problem can be de-

[1]Modified version of Table 8.1: The three agreement problems presented by Singhal and Shivaratri [11].

Problem	Byzantine Agreement	Consensus	Interactive Consistency
Who proposes initial value	One processor	All processors	All processors
Final agreement	Single value	Single value	A set of values

TABLE I
AGREEMENT PROBLEMS.[1]

rived from solutions to the Byzantine agreement problem, which simplifies implementation and promotes reusability.

B.2 Solutions to the Byzantine Agreement Problem

The Byzantine agreement problem, initially defined and solved by Lamport, Shostak, and Pease [12], [13], has been extensively studied over the years. Every protocol that attempts to solve this problem guarantees that all nonfaulty processors agree in the same value, and that if the processor who initiates the value is nonfaulty, the common value agreed by all processors should be the initial value that was proposed. Moreover, solutions to agreement problems have in common the following system model [11]:

- There are n processors in the system and at most f of them are faulty.
- Processors communicate directly to other processors by message passing. Therefore, a logically fully-connected system is assumed.
- A receiving processor always knows the identity of the sending processor.
- The communication medium is reliable[2], but processes may fail arbitrary.

Another important assumption is a synchronous model of computation, based on rounds. A *round* refers to a computational step where a process receives a message (sent in the previous step), performs the required computations, and sends the resulting messages to other processes (that will be received in the next round). Processes have knowledge about the messages they expect to receive in a round.

Different solutions for the Byzantine agreement problem deal with the trade-off between message complexity and number of rounds in different ways. Garay and Moses [14] listed several solutions presented though the years and compared them in terms of n (total number of processes), required number of rounds for agreement, communication, and computation.

C. Message Digest: SHA-1

A message digest (also known as a hash) is a one-way function that takes an input message and produces an output. The one-way property is what makes a message digest function cryptographically secure. It should be computational infeasible to determine a message, given its message digest, and, similarly, it should be impossible to find two messages that produce the same message digest [15].

There are several algorithms to compute message digests: MD2, MD4, MD5, and SHA, among the well-known ones. In particular, SHA (Secure Hash Algorithm) is a specification for computing a condensed representation of a message or a data file proposed by the National Institute of Standards and Technology (NIST) as part of the Secure Hash Standard (SHS) [16]. There are several versions of the Secure Hash Algorithm: SHA-1, SHA-256, SHA-384, and SHA-512 [17]. They mainly differ in the length of the message digest, ranging from 160 to 512 bits, depending on the algorithm. Nevertheless, currently, the only FIPS-approved [3] algorithm for generating message digests is SHA-1. In addition to that, SHA-1 is the algorithm specified in the Digital Signature Standard (FIPS PUB 186-2) to generate the condensed version of the message to be signed [18] and is the U.S. government's standard hash function.

III. PROTOTYPE IMPLEMENTATION

A. Need for Agreement in CRLib

In order to provide an application with the ability to sustain operation and dynamically restore the level of assurance in system function during attack, once a programmer has set up the communication structure of the application and the level of resiliency (i.e., number of replicated threads), CRLib supplies means for dynamic reconfiguration and replication of threads. Dynamic reconfiguration of groups of replicated threads involves solving critical issues such as describing and managing the group, detecting a compromise, and ensuring valid program state and communication structure [3]. CRLib addresses those problems through the implementation of three protocols: the group membership protocol, the liveness checking and recovery protocol, and the flow control protocol. The membership protocol provides mechanisms to create threads and to add (join operation) or withdraw (leave operation) threads from a group. The liveness checking and recovery protocol implements an interface to application specific routines for detecting a compromise and the recovery mechanism. The flow control protocol provides communication abstraction based on ordered delivery multicast operations. Details of communications are hidden to the programmer, as it is shown in the Figure 1 where replicated threads multicast messages to threads in the receiving group, but to the application, only one high level message is sent.

[2]The same authors report that, recently, agreement problems have been studied under failures of the communication links only and under both communication and process failures.

[3]NIST cryptographic standards are specified in Federal Information Processing Standards (FIPS) Publications. The term FIPS-approved indicates something (e.g., a cryptographic algorithm) that is either a) specified in a FIPS or b) adopted in a FIPS and specified either in an appendix to the FIPS or in a document referenced by the FIPS.

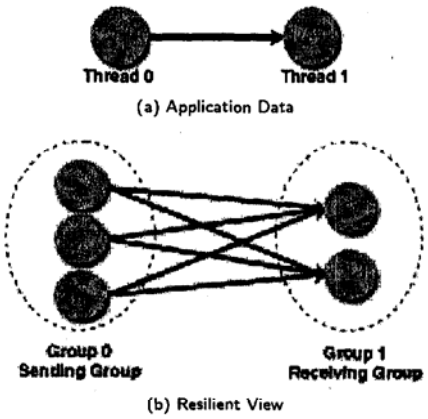


Fig. 1. Group Channel Implementation.

In particular, the flow control protocol ensures reliable delivery of messages in the sense that each receiving thread will get one or more copies of the same message. Figure 2 illustrates this situation. Threads in the sending group multicast their messages to threads in the receiving group. Consequently, any thread there could received the same message several times, as many times as the level of resiliency of the sending group, resiliency level 3 in the example. The receiving thread discards duplicated messages, reorders them, and then sends them to the application level thread.

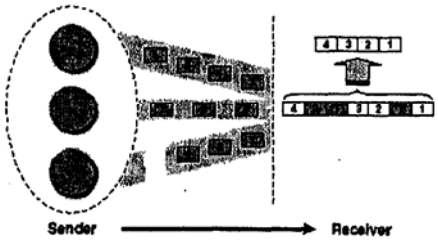


Fig. 2. Flow Control Mechanism.

However, we do not assume an underlying reliable multicast mechanism. Our multicast is implemented within the user process, and at that level is implemented over unreliable unicast mechanisms, making it susceptible to compromise. Because when using CRLib the multicast mechanism is under the control of the sender, it is possible for the sender to achieve an asymmetric or Byzantine fault by dividing the multicast groups at the unicast level.

In addition to that, because replicated threads can be located in different systems, intermediate computations can differ. This, for certain scientific requiring precise numerical values, could produce erroneous results or cause divergence or disagreement on a final value. Figure 3 shows such situation.

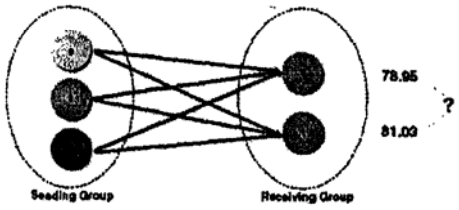


Fig. 3. Possible Scenario where Replicas of Same Thread Receive Different Messages, Leading to Inconsistent Values.

To achieve consistency in the presence of faults (considering each replicated thread in the sending group as a potentially arbitrary faulty thread), receiving threads should exchange their values with other peer threads and relay the values received from them several times. This process of isolating the effects of faulty threads is exactly that of reaching a consensus.

B. Group Communication Cases in CRLib

Based on the resiliency (replication level) of both the sending and the receiving group, there are four different cases of group communication in CRLib. Some of these cases correspond to the well-known agreement problems described in Table I, which allow presenting a solution for the communication problems based on agreement protocols.

B.1 Case 1: Both Resiliency of the Sending Group and Resiliency of the Receiving Group are equal to 1

This is the simplest case. The single copy of the thread in the sending group sends messages to the single copy of the thread in the receiving group (see Figure 4). Since communication is one to one, its solution is trivial: there is nothing to agree on. The single message received is considered valid.

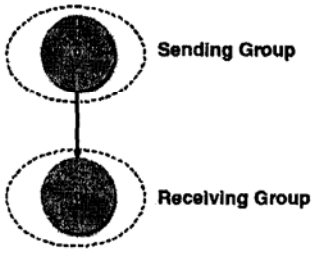


Fig. 4. Case 1: Sending group resiliency == 1 and Receiving group resiliency == 1.

B.2 Case 2: Resiliency of the Sending Group is equal to 1, but Resiliency of the Receiving Group is greater then 1

The single copy of the thread in the sending group broadcasts messages to each replicated thread in the receiving group (see Figure 5). Because there is no guarantee that the sending thread is not faulty (and it can therefore be sending spurious messages), threads at the receiving end need to *agree* in the value received. This case corresponds to the Byzantine Agreement problem.

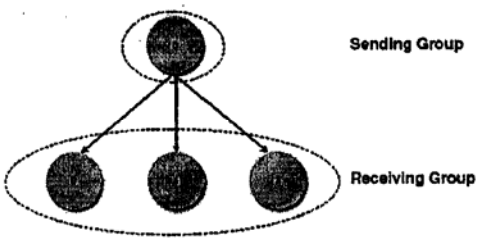


Fig. 5. Case 2: Sending group resiliency == 1 and Receiving group resiliency > 1.

B.3 Case 3: Resiliency of the Sending Group is greater than 1, and Resiliency of the Receiving Group is equal to 1

In this case, the single copy of the thread in the receiving group receives as many messages as the number of replicated threads (level of resiliency) in the sending group, as shown in Figure 6. Although no agreement is needed, the receiving thread still needs to decide which of the message values is valid. For that, a simple voting algorithm satisfies the need.

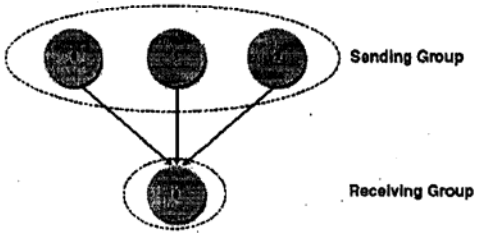


Fig. 6. Case 3: Sending group resiliency > 1 and Receiving group resiliency == 1.

B.4 Case 4: Resiliency of both, Sending Group and Receiving Group is greater than 1

This is the most complex case. Each replicated thread in the sending group broadcasts a message to each thread in the receiving group, as it is illustrated in Figure 7. Thus, threads in the receiving end need to reach agreement on each message received. Therefore, several rounds of agreement are needed— as many as the resiliency level of the sending group (which is three in the example). In addition to that, once replicas in the receiving group have agreed on the values received from each thread of the sending group, they need to decide what message is valid. For that, if each of them executes a voting algorithm they can easily discard any spurious message sent by a faulty thread.

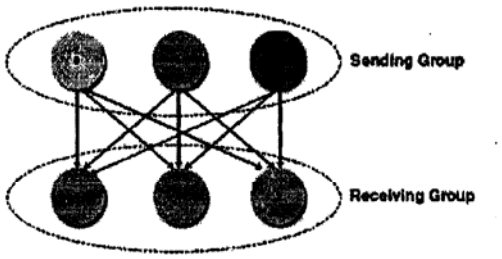


Fig. 7. Case 4: Sending group resiliency > 1 and Receiving group resiliency > 1.

C. Solution Implementation Details

The solution presented in this paper aims to solve the need for reaching agreement and consensus in some of the communication cases of CRLib efficiently and without causing too much overhead. A similar algorithm to the one presented by Lamport, Shostak, and Pease [12] was used to reach agreement, but instead of using the original application messages the described solution uses hashes of those messages. This algorithm is conservative in comparison with more recent algorithms, but makes not assumptions about the type of malicious faults, which is exactly our assumption.

We compute message digests using SHA-1 to reduce the communication overheard when executing the agreement protocol, which is particularly important considering that CRLib can be used for distributed applications that deal which extremely large messages. However, this particular feature is susceptible to the problem of inconsistent values when dealing with heterogeneous machines, and even in some homogeneous cases: floating-point computations can result in slightly differing values. Because of the properties of the hashing algorithm, a one-bit difference between two inputs will produce differing hashes, and there is no correspondence between the difference distance between the inputs and the difference distance between the hashes. Then, reaching agreement or consensus would be almost impossible under such circumstances. To solve this problem, we developed the notation of fuzzy agreement. The fuzziness is eliminated right before computing the message digest. A user-defined application-dependent function normalizes the message to a value within a valid range determined by the user and then, the message is hashed: A simple

example for floating point is to round the input to a precision that is consistently represented across machines. In that way, the probability of obtaining identical hashes for nearly identical messages is much higher.

C.1 Agreement

The implemented algorithm corresponds to the solution with oral messages presented by Lamport, Shostak, and Pease [12]. Assuming that n is the total number of replicas and f the number of faulty replicas, there must be at least $3f + 1$ replicas to cope with f. The characteristics of oral messages are given by the following assumptions:

- Every message that is sent is delivered correctly.
- The receiver replica knows who sent the message.
- The absence of a message can be detected.

Furthermore, the algorithm assumes a function *majority* that takes a set of values $v_1, \ldots, v_{n-1}$ and determines a v, which would usually be the v_i repeated the greatest number of times, but actually it could be the median or any other value depending on the application. The *majority* function is a user-defined function in the implemented solution.

Figure 8 shows in detail the Lamport-Shostak-Pease Oral Messages Algorithm for replicated threads, where the initiator thread is the replica initiating the communication.

Algorithm OM(0)
(1) The initiator thread sends its value to every replicated thread.
(2) Each replica uses the value it receives from the initiator thread, or uses a default value if it receives no value.

Algorithm OM(f), f > 0
(1) The initiator thread sends its value to every replicated thread.
(2) For each i, let v_i be the value replica i receives from the initiator thread, or else be a default value if it receives no value. Replica i acts as the initiator thread in Algorithm OM(f - 1) to send the value v_i to each of the n - 2 other replicas.
(3) For each i, and each $j \neq i$, let v_j be the value replica i received from replica j in step (2) (using Algorithm OM(f - 1)), or else a default value if it received no such value. Replica i uses the value *majority* $(v_1, \ldots, v_{n-1})$.

Fig. 8. The Lamport-Shostak-Pease Oral Messages Algorithm.

C.2 Extending Implementation of Group Semantics

Groups of replicated thread in CRLib are open. That is, other threads outside the group may send messages to the group. Initially, CRLib provided only communication mechanisms that allowed replicas from one group to send messages to replicas in a different group but not among peer replicas of the same group. To reach agreement, member threads of the same group need to relay messages among each other. Therefore, additional internal group communication mechanisms were implemented allowing member threads, within the same group, to multicast to themselves.

C.3 SHA-1

The implementation of the SHA-1 algorithm, in accordance with the Secure Hash Standard (FIPS PUB 180-1), is an independent module added to the CRLib for two purposes: one is to produce a unique constant-size version of the message (160 bits) for use in agreement, and the other to facilitate implementation of alternative solutions that require signed messages.

C.4 Fuzziness

The user can provide an application-dependent function that CRLib will use to round the message to a certain threshold defined by the user. This furnishes a means of eliminating the fuzziness that could lead to the impossibility of reaching agreement in a value other than the default one, because of the fact that the resulting message digests are completely different. The performance of this function will depend on the implementation and on the size of the messages.

IV. Experimentation

Singhal and Shivaratri [11] report *time*, *message traffic*, and *storage overhead* as the most commonly used metrics for measuring performance of agreement protocols. Time refers to the time taken to reach an agreement under a particular protocol, that is, the number of rounds needed to reach agreement. Message traffic is usually measured in two ways: as the number of messages exchanged or as the total number of bits exchanged to reach agreement. Storage overhead measures the number of bytes of data stored at the processors when executing the agreement protocol.

The implementation of our solution to agreement problems reduces the number of bits exchanged and stored when doing agreement through a constant 160-bit message size, a significant improvement considering that many distributed applications deal with large messages. The time required to reach agreement of the solution is the same time of the Lamport-Shostak-Pease solution, $f + 1$ rounds as well as the number of messages exchanged, $O(n^f)$. However, our solution also minimizes communication and storage overhead, because the number of bits exchanged and stored is much less in comparison to traditional solutions.

A. Basic Test Application

The basic test application is a very simple program that transfers arrays of characters back and forth. There are no computations performed over those string values. The purpose of such a minimalistic application design was to simplify the measurement of communication overhead. These results closely track those achieved with more complex applications, so we present the minimalistic application for the sake of simplicity.

Results of tests with the number of replicas, n, equal to 4, 7, and 10 (for which a number of faulty threads, f, of 1, 2, and 3, respectively, were assumed) indicate different degrees of overhead depending on the communication case.

Figure 9 compares the performance of the application with different message sizes when a single thread multicasts its value to several replicated threads (see communication

case 2, illustrated in Figure 5). For this particular case, the overhead corresponds to the communication overhead when exchanging hashes in order to reach agreement plus the computation overhead of hashing the message itself. Observed results show that the performance degradation is never greater than 48% when compared to the same application run without any agreement.

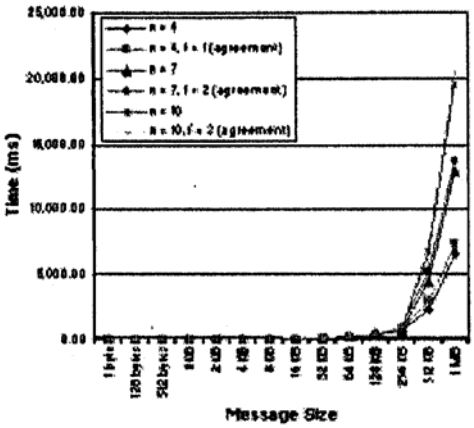


Fig. 9. Execution Time of Basic Application when Reaching Byzantine Agreement (one sending replica, many receiving replicas) with n Total Replicas of which f are Faulty.

Similarly, Figure 10 shows the execution times of the application, with the same message sizes, but when many replicated threads multicast their values to a single thread (see Figure 6). Since no agreement is needed, just a simple voting to select one of the messages, the overhead in this case is minimal and not a product of the communication but of the hashing, rounding, and voting.

On the other hand, Figure 11 illustrates the performance with the same number of replicas and the same message sizes in the most complicated case of communication: when several threads of one group multicast to several other threads in some other group. This is the consensus case shown in Figure 7. Overhead in this case is a sum of the execution time of the hashing function, the communication time of reaching consensus, and the final voting to select which of the agreed messages is the valid one. In particular, this communication overhead is much higher than in other cases because several rounds of agreement need to be performed.

In our approach, there is a tradeoff of speed vs. accuracy when compared to sending the data for agreement rather than the hash in the case of consensus agreement. Because the number of rounds and messages sent is invariant with respect to sending hashes or full messages for agreeemnt, the overhead of sending full messages is roughly equal to the ratio between the hash size and the size of the original

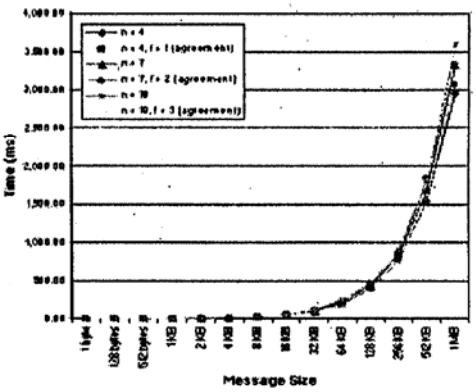


Fig. 10. Execution Time of Basic Application when Voting (many sending replicas, one receiving replicas) with n Total Replicas of which f are Faulty.

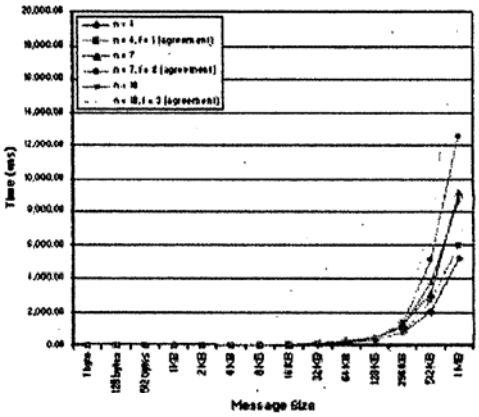


Fig. 11. Execution Time of Basic Application when Reaching Consensus (many sending replicas, many receiving replicas) with n Total Replicas of which f are Faulty.

message. In the case of the SHA-1 algorithm, this overhead for a message M is just $\frac{sizeof(M)-20}{20}$, which scales linearly with the size of the message. For 128-byte messages, the communication load is 5 times higher than with our method; for 4-kilobyte messages, it is more than 200 times higher, and for 1-megabyte messages, more than 52,000 times higher.

Consider that for a replication level of 10, allowing for three failures, the hashed messages took 34 milliseconds for 4K messages, 306 milliseconds for 64K messages, and 19 seconds for 1M messages. If we sent the complete messages, this agreement would take in the neighborhood of 6 seconds for 4K messages, one minute for 64K messages,

and more than an hour for 1-megabyte messages. Considering that scientific calculations typically exchange data on each iteration step and run for thousands to millions of iterations, this overhead can quickly come to dominate the computation time, making it infeasible to run simulations that send large messages.

The cost of this efficiency is the risk that we may refuse to agree in a situation where an alternative method might find agreement. In a contrived example, we have four processes agreeing on a two-element vector, and the rounding function rounds to the nearest integer before computing the hash. With the four vectors as follows:

$v_1 = [0.49 \quad 1.51] \quad v_2 = [0.50 \quad 1.50]$
$v_3 = [0.51 \quad 1.52] \quad v_4 = [0.52 \quad 1.49]$

which, after rounding, become:

$v_1 = [0 \quad 2] \quad v_2 = [1 \quad 2]$
$v_3 = [1 \quad 2] \quad v_4 = [1 \quad 1]$

we will be unable to agree after hashing, because no three hashes will match. However, if we instead agreed on the individual components of the vectors, we would see that three out of the four vectors match on each of the elements, and could agree upon [1 2].

In actual use with three appliction programs, we have not found any problem with accuracy in agreement. We believe the ability to attempt agreement on much larger messages than would otherwise be practicable outweighs the possibility of disagreement on a small subset of the possible messages.

V. Conclusions and Future Work

A solution that provides fuzzy agreement, guaranteeing consistency of the data in the presence of arbitrary faults, when doing group communication in CRLib was implemented and shown to work. Such an implementation is a significant improvement to the library since it makes it much more reliable and ensures data integrity. An innovative approach of agreeing on hash values of the messages, computed using SHA-1, reduces the total number of bits exchanged to reach an agreement. Therefore, the expected communication overhead was minimized. Furthermore, means to eliminate the fuzziness present when doing computations in heterogeneous environments are provided, so digests of similar messages are the same and an agreement could be reached.

A basic test application was developed to measure the overhead in performance when dealing with different communication cases and a variety of message sizes. Results of these experiments indicate that due to the number of rounds of agreement needed to reach consensus, communication is the dominant form of overhead, and our approach is up to 50,000 times more efficient for large messages because of the small constant size of the SHA-1 hash. In addition to that, tests over a real-world application using the CRLib when supporting both fail-stop and arbitrary failure models suggested that the performance overhead was acceptable in cases when any kind of failures can occur.

Our future work will include the use of signed messages in the agreement protocol, thereby constraining on n to be greater than $2m$, rather than greater than $3m$ with unsigned messages.

References

[1] M. C. Libicki, "What is information warfare?." ACIS Paper 3, National Defense University, Fort McNair, D.C., August, 1995. Retrieved on May 05, 2002 from the World Wide Web: http://www.ndu.edu/inss/actpubs/act003/a003.html.

[2] S. Jajodia, C. D. McCollum, and P. Ammann, "Trusted recovery," *Communications of the ACM*, vol. 42, pp. 71–75, July, 1999.

[3] J. Lee, S. Chapin, and S. Taylor, "Computational resiliency," *Journal of Quality and Reliability Engineering International*, vol. 18, pp. 1–15, March 2002.

[4] J. Lee, *Computational Resiliency: Reliable Heterogeneous Applications*. PhD thesis, Syracuse University, 2001.

[5] J. Lee and S. Taylor, "Advances in computational resiliency," in *2001 IEEE Aerospace Conference Proceedings* (2001, ed.), vol. 6, (Big Sky, MT).

[6] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and design*. Harlow, England: Addison-Wesley, 3rd ed., 2001.

[7] M. H. Azadmanesh and R. M. Kieckhafer, "Exploiting omissive faults in synchronous approximate agreement," *IEEE Transactions on Computers*, vol. 49, pp. 1031–1042, October, 2000.

[8] F. J. Meyer and D. K. Pradhan, "Consensus with dual failure modes," in *Proceeding of the 17th Fault-tolerant Computing Symposium*, pp. 48–54, July, 1987.

[9] P. M. Thambidurai and Y.-K. Park, "Interactive consistency with multiple failures modes," in *Proceedings of the Seventh Reliable Distributed Systems Symposium*, October, 1988.

[10] M. H. Azadmanesh and R. M. Kieckhafer, "New hybrid fault models for asynchronous approximate agreement," *IEEE Transactions in Computers*, vol. 45, pp. 439–449, April, 1996.

[11] M. Singhal and N. G. Shivaratri, *Advanced Concepts in Operating Systems: Distributed, database, and multiprocessor operating systems*. McGraw-Hill, 1994.

[12] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July, 1982.

[13] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, pp. 228–234, April, 1980.

[14] J. Garay and Y. Moses, "Fully polynomial byzantine agreement in t + 1 rounds," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, (San Diego, CA), pp. 31–41, 1993.

[15] C. Kaufman, R. Perlman, and M. Speciner, *Network Security: Private communication in a public world*. PTR Prentice Hall, 1995.

[16] "Secure hash standard." Federal Information Processing Standard Publication (FIPS PUB 180-1), U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, April 17, 1995. Retrieved on January 31, 2001 from the World Wide Web: http://csrc.nist.gov/publications/fips/fips180-1/fips180-1.pdf.

[17] "Secure hash standard." Federal Information Processing Standard Publication (FIPS PUB 180-2), U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2001, Retrieved on June 01, 2001 from the World Wide Web: http://csrc.nist.gov/encryption/shs/dfips-180-2.pdf.

[18] "Digital signature standard (dss)." Federal Information Processing Standard Publication (FIPS PUB 186-2), U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, January 27, 2000.

Appendix D

Published in Journal of Quality and Reliability Engineering International, John Wiley & Sons, Ltd., May/June, 2002

COMPUTATIONAL RESILIENCY[1]

Joohan Lee, Steve J. Chapin, Stephen Taylor

3-114 CST building, Syracuse University
Syracuse, NY 13244
Tel: 315-443-2129, Fax: 315-443-1126, jlee@ecs.syr.edu

[1]This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) under contract N66001-99-1-8922 and the Air Force Research Laboratory (AFRL), Information Warfare Directorate, Rome Laboratory, NY.

SUMMARY

The notion of *computational resiliency* refers to the ability of a distributed application to tolerate intrusion when under information warfare (IW) attack. This technology seeks an active strengthening of a military mission, rather than protecting its network infrastructure using static defensive measures such as network security, intrusion sensors, and firewalls. Computational resiliency involves the dynamic use of replication, guided by mission policy, to achieve intrusion tolerance so that even undetected attacks do not cause mission failure, however, it goes further to dynamically regenerate replication in response to an IW attack, allowing the level of system assurance to be restored and maintained. Replicated structures are protected through several techniques such as camouflage, dispersion, and layered security policy. This paper describes a prototype concurrent programming technology that we have developed to support computational resiliency and describes how the library has been applied in two prototypical applications.

KEY WORDS: Computational resiliency; fault tolerance; distributed computing; multithreading; information warfare, network security

INTRODUCTION

The ability to tolerate failures and attacks is desirable for many real-time distributed applications, especially mission-critical applications such as command and control, surveillance, and electronic commerce. Even long-running parallel applications require fault tolerance to avoid a complete restart of the program in the presence of failure. Generally, the fault-tolerant mechanisms to achieve this robust operation are based on some form of replication of critical information and resources. While this often provides graceful degradation of system performance, it is clearly not sufficient to aggressively recover assured operation.

In this paper, we investigate an alternative model of distributed computation termed *computational resiliency* [1, 2]. This model provides a distributed system with the ability to sustain operation and dynamically restore the level of assurance in system function in the presence of information warfare attacks by combining real-time attack assessment with transparent and automatic reconfiguration and recovery. It assures that operational readiness is eventually restored subject only to the constraints imposed by available resources.

To visualize how these concepts might operate, consider a distributed application as analogous to an apartment complex inhabited by a new strain of roach (process/thread)[2]. The roaches are highly resilient: you can stamp on them, spray them, strike them with a broom but you never kill them all or prevent them from their goal of finding food (resources). To foil your eradication efforts, they use several techniques: they are *highly mobile* moving from one place in the apartment complex (network) to another with speed

[2] Thanks to Cathy McCullum for providing this analogy.

and agility. They continually *replicate* to ensure that it is not possible to kill them all. They *sense* their environment (attack assessment) to obtain clues that mobility is necessary: if a light is turned on, they scurry away in all directions to hide behind cupboards in places of *known safety* (secure network zones). If a new roach killer is invented they *learn* from it, and *adapt* their behavior to compensate. However, this new strain is particularly aggressive and seeks to live in the daylight (wide-area operation): thus it adopts techniques for camouflage as a form of protection and disinformation.

To realize this resilient computing model, we have developed an application-independent, distributed programming middleware library. The developed library provides Application Programming Interface (API)s for concurrent programming and representing the user's requirements for reliability. Details of the communication protocols required to achieve on-the-fly replication and reconfiguration are hidden from the users. The library also ensures that no communication is lost, that the integrity of the process state is maintained, and that where possible locality of communication is preserved. It operates on a broad variety of heterogeneous networked architectures that include shared-memory multiprocessors and clusters of workstations.

We have applied the developed technology to two prototype distributed applications: a hyper-spectral remote sensor and fluid dynamics to investigate the performance issues associated with the technology. In this paper, we describe the prototype implementation of the technology and show how those applications can benefit from it. We also provide performance measurements to quantify the associated overhead of resiliency over a distributed system consisting of 32, dual-processor, shared-memory multiprocessors connected with switched fast-Ethernet technology.

RELATED WORK

Fault-tolerance techniques can be implemented using only hardware or software, and some techniques need both of them. Hardware techniques require higher cost but give better performance. On the other hand, software techniques are more flexible for future modification. Here we are concerned primarily with software-based techniques that can be applied to distributed real-time applications. Most of the techniques developed to date are based on notion of *process replication* to provide high levels of system availability [3]. However, the use of replication introduces additional problems such as the need to maintain consistency between replicas, detect the failure of a compromised process, and transparently recover system function.

Most distributed systems use software-based replication techniques that can be divided into two general categories based on *passive* [4] or *active* [5] replication to provide fault tolerance. In passive replication [4], there is a single primary server and all other replicas are maintained purely as backups. Passive replication involves less redundant processing and is less costly, however, this method is slow to transfer control to a backup in the event of failure; this can lead to significant degradation in system response. Active replication [5] has no centralized control. Any viable replica may receive a message from a client and collectively the replicas maintain message ordering and atomicity. This approach is attractive for real-time systems because it is relatively fast to transfer control in the event of failure although it requires more computing resources [6].

To implement replication it is useful to organize processes into *groups* and provide communication mechanisms between groups. The processes in a group cooperate to provide a single service and each group is viewed as a single logical entity hiding its

internal structure from other groups [7]. The processes use multicast communication primitives that guarantee every process in the group receives the same messages in the same order to maintain and a consistent process state. The group concept has been extended to many fault-tolerant distributed systems such as Isis [8], Horus [9], Transis [10], Totem [11], and Ameoba [12]. Although not used for fault tolerance, the process group has also been used widely as a concurrent programming paradigm through libraries such as PVM [13] and MPI [14].

A useful taxonomy of database recovery techniques for information warfare has been developed by Jajodia [15]. Cold-start recovery involves a complete restart in the event of a severe attack. Warm-start involves non-transparent but automated recovery. Hot-start techniques are by far the more sophisticated and provide transparent recovery. These techniques operate through a combination of implementation techniques that include checkpoints and intelligent analysis of the effects of attack queries [15]. Checkpointing generally requires more time to recover than process groups since it involves restoring previous state information from a stable repository such as hard disk and starting a new process. Checkpointing mechanisms can sometimes be used transparently and a variety of techniques have been developed to reduce the associated overheads [16, 17, 18, 19].

COMPUTATIONAL RESILIENCY

To tolerate information warfare attacks, applications may choose to statically replicate mission critical threads, thereby forming *thread groups*, as shown in Figure 1. Each thread in a group is allocated to a different computational resource so as to sustain operation. This provides a graceful path of performance degradation to the point of

failure. Unfortunately, it is not resilient in that it does not *assure* continued operation of the system when sufficient resources are available elsewhere in the network. In any realistic system, there will never be sufficient resources to replicate all threads, therefore some policy-based methods for controlling replication are required. In addition, resources may become available, or unavailable, dynamically, during the course of a conflict.

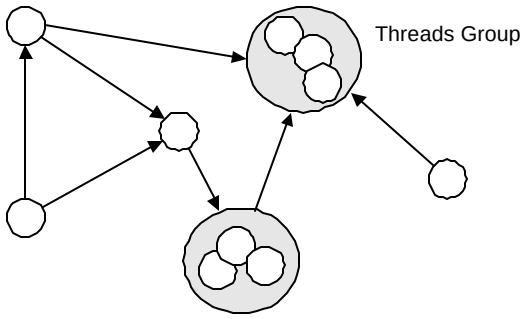


Figure 1. Replication of Threads

An alternative approach is to *dynamically recreate the level of thread replication* in the face of attack. This assures that operational readiness is eventually restored, subject only to the constraints imposed by the time-dependent availability of resources. Obviously to be successful, the replacement thread must be mapped to an alternative location in the network with sufficient resources. Protocols are required to dynamically reconfigure communication between residual thread groups and newly created replicas. These protocols deal with race conditions inherent in the reconfiguration process, ensure that no communication is lost, that the integrity of state is maintained, and that where possible locality of communication is preserved.

Figure 2 shows how resiliency is layered into a distributed application. The application programmer simply describes the required thread structure and states the level of

resiliency for each crucial thread. In the diagram there are three threads, the first and second are resilient to level three, while the third is resilient to level two. Communication between threads at the application level is replaced by group communication at the resilient level. Threads are subsequently mapped, through indexing, to appropriate processors such that replicas in a single group are placed in different processors at the architectural level.

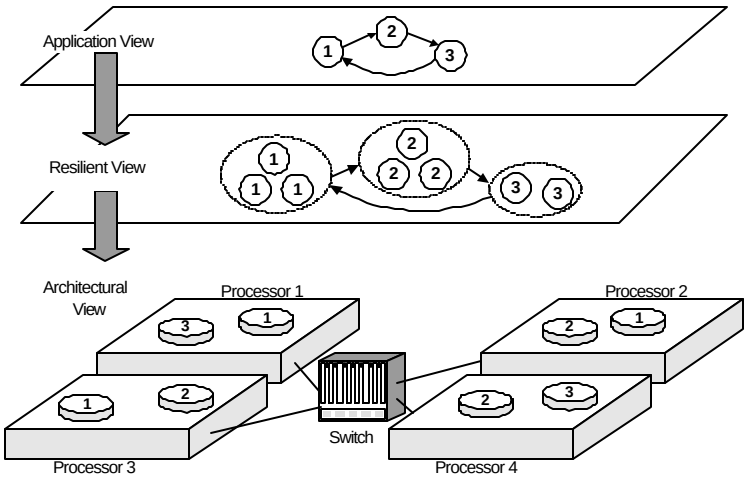


Figure 2. Computational resiliency using a Cluster of Multiprocessors

Figure 3 compares the fault-tolerant model of computation (dashed line) with computational resiliency (solid line). In a fault-tolerant implementation, as threads are compromised, graceful degradation occurs and eventually, when no replicas are available, the application is unable to proceed.

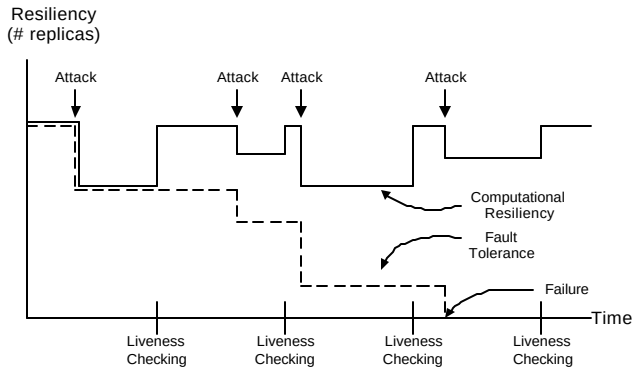


Figure 3. Fault-tolerance vs. Computational Resiliency

Using resiliency, periodic liveness checks are performed. These checks are *not* designed to detect an IW attack, but rather, they seek to determine if an application is not performing as expected. If an application thread is detected as compromised during a liveness check, it will be destroyed and replaced using the uncompromised residual members of the group. This hot-start recovery mechanism [13] ensures that the newly recreated thread begins execution from the most recent state rather than a state where the compromise occurred. No message logging or intermediate state is saved either in stable storage such as a hard disk, or at a remote server. Therefore, network file system failure does not affect robustness.

PROTOTYPE IMPLEMENTATION

To provide highly mobile threads with the ability to reconfigure and replicate in a heterogeneous computing environment, it is necessary to have an explicit representation of the *communication structure* used by the application. We have developed a concurrent programming library that provides this basic functionality [1, 2, 18, 19, 20, 21]. The library is portable to a wide range of platforms, from distributed-memory multicomputers

to networks of workstations, PC's and multiprocessors. It provides a *mobile thread* abstraction in which threads may move between processors to accommodate for changes in resource requirements (e.g. processor speed, memory, bandwidth).

Distributed applications are composed of a collection of threads that communicate and synchronize either through shared memory or by sending messages. Each thread has an associated *state,* which is operated on by application specific routines e.g. in a remote sensing application this may involve matrix algebra for image manipulation. A thread also has a machine independent description of its communication structure. In general these systems are *reactive* [24] in that the important transitions between data states occur at the receipt of messages. This provides a natural mechanism to synchronize each thread, detect an information warfare attack, and initiate appropriate recovery.

To dynamically recover replication, a mechanism is required to recreate a compromised thread with the appropriate communication structure at a new location in the network. This mechanism is implemented by replicating a residual thread from the compromised group and subsequently moving the new thread to its desired location. Unfortunately, it may not be efficient, either because of memory disparities or because of thread *granularity* (i.e. ratio of computation to communication) to move the new thread directly. Thus additional mechanisms are needed to allow thread granularity to be increased, by merging, or decreased, by splitting, the associated computation. Armed with these basic techniques: thread *move*, *merge* and *split*, it possible to build concepts for resource management [22]. There exists no general solution to the resource management problem, thus each application must employ an appropriate technique [25]. For simplicity, in this paper a Manager-Worker approach is used to demonstrate the ideas [26].

The crucial issues associated with use of a dynamically reconfigurable group of replicated threads include describing and managing the group, detecting a compromise, and ensuring valid program state and communication structure. From a programming perspective we seek to hide the implementation details associated with these issues in a programming library and so relieve the application programmer from the complexities associated with resiliency. A programmer may simply specify the level of resiliency (i.e. number of thread replicas required) when initially spawning a thread. This level of resiliency will then be maintained automatically throughout the runtime of the computation provided that there are sufficient resources. Resiliency will gracefully degrade when resources are stretched beyond their capacity. The programming library implements three protocols that address these crucial issues by providing group membership, liveness checking and recovery, and flow control. We use two prototypical applications to demonstrate how those protocols are used and quantify the associated performance.

Membership Protocol

The membership protocol provides mechanisms to *create* threads and cause them to *join* or *leave* a group. At the beginning of program execution, groups are constructed by creating replicas and causing each to join the group. During failure and recovery, when a thread is compromised, it is forced to leave its group. A new replica is then created that joins the group to take its place. Groups are numbered for addressing purposes and the organization of communication is keyed to this numbering. Programmers follow a

standard Application Programming Interface (API) that allows them to specify the required resiliency for a thread and create communication channels between groups:

```
int  group_create(thread_fn, resiliency)

channel_create(group1, group2)
```

Figure 4 shows abstractly how these functions are used to implement the process structure shown in Figure 2. Three groups are created (1,2,3), each designated by an appropriate unique identifier (g1,g2,g3). When the first group is created the programmer specifies resiliency of three, meaning three replicated threads in the group g1 (1). Similarly, the second thread has three replicas and the third has two. The mapping of threads in this example occurs abstractly through the @ notation. After groups are created, architecturally independent communication channels are created between groups (4,5,6). Communication between threads that are not replicated, i.e. resiliency 1, involves no overhead associated with group representation.

```
main () {
   g1 = group_create (thread1_fn, 3) @ C1, C2, C3   // 1
   g2 = group_create (thread2_fn, 3) @ C2, C3, C4   // 2
   g3 = group_create (thread3_fn, 2) @ C1, C4       // 3
   channel_create (g1, g2)                          // 4
   channel_create (g2, g3)                          // 5
   channel_create (g3, g1)                          // 6
}
```

Figure 4. Abstract Code for Figure 2

Liveness Checking Protocol

The liveness checking and recovery protocol provides an interface to application specific routines for detecting a compromise and implements the recovery mechanism. The

frequency of liveness checking is determined by the programmer and is application dependent.

Liveness checking uses the underlying mechanisms for thread movement to recreate compromised threads and reconfigure group communication. When a liveness check occurs, the members of each group communicate, and one of the uncompromised threads is selected as the group leader. The leader then coordinates dynamic replication and changes to the group communication structure to both exclude compromised threads and include the new replicas. Bounds on latency and timeouts are used to circumvent compromised threads that simply fail to respond during the protocol. The application programmer is simply required to determine at what point a liveness check is to be performed.

In our implementation, we provide two types of liveness checking, synchronous liveness checking and asynchronous liveness checking. In synchronous liveness checking, liveness checking is performed globally across the application such that all the threads in the system are involved in this operation. It requires a global synchronization point across the distributed threads. This type of liveness checking is appropriate for those applications with Single Instruction Multiple Data (SIMD) style parallelism where each processor executes the same program and such a global synchronization point can be determined easily, for example, at the end of each iteration of a loop. Assuming that the workload is evenly distributed over the threads, there wouldn't be significant overhead due to the global synchronization. However, it may not be applicable for a certain class of applications. It is inefficient and costly for irregular parallel applications where workload of each thread can vary such that the threads with more workload can hold the other

threads unnecessarily. Some applications may want to perform the liveness checking more frequently on some part of groups than the others. Moreover, such a global synchronization point may not exist in some applications especially with Multiple Instruction Multiple Data (MIMD) parallelism where each processor executes different programs. For that purpose, the library provides another type of liveness checking, asynchronous liveness checking. It allows only the threads in the same group to perform the liveness checking regardless of the other groups and the operation is not visible to the others. Therefore, it can be used for the applications where the global liveness checking is inapplicable. However, per group basis liveness checking causes some issues regarding consistency. Other threads are not aware of the liveness checking going on in a certain group of threads and their states keep changing as the computation proceeds. Even they can send messages to that group while its group members are performing liveness checking which could be lost especially if they are reconstructing their computation and communication structures due to the failures. Figure 5, illustrates the problem of possible message loss during liveness checking. In Figure 5(a), both group 1 and group 2 perform synchronous liveness checking. One of the members in group 1 crashed and a new replica is created during the liveness checking period represented as lined rectangle. In synchronous liveness checking the threads don't proceed until all the threads finish liveness checking; therefore, there are no messages in transit during the liveness checking period. However, in asynchronous liveness checking, Figure 5(b), only group 1 performs synchronous liveness checking. It shows that the messages sent from group 2 can be missed for the new member in group 1 with asynchronous liveness checking since the

sender is not aware of the new thread in group 1 until the liveness checking is finished and does not retransmit the missed messages.

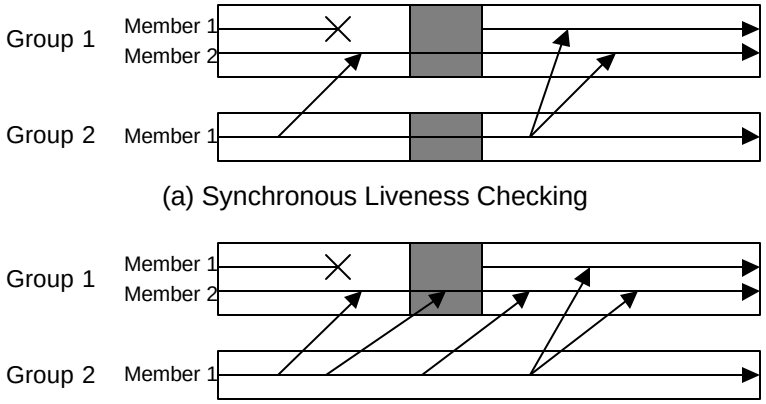


Figure 5. Issues with Messages in Transit

To deal with messages in transit during the liveness checking, asynchronous liveness checking uses message forwarding. To guarantee the delivery of the messages that could be missed for the new thread, the other members in the group 1 forward the messages to it until it can receive the messages from the original sender correctly. While asynchronous liveness checking can be used for wider range of applications, it causes more overhead in the recovery process due to the message forwarding.

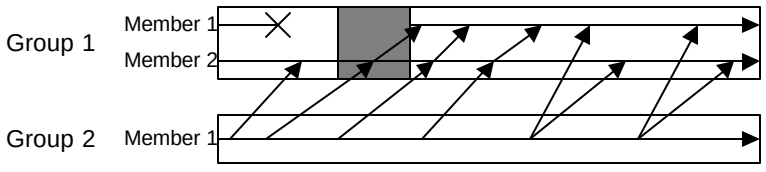


Figure 6. Message Forwarding

Assume that the process structure in Figure 2 is used to simply circulate a token among the threads. The first thread is responsible for injecting the token (1,2). The basic action of each thread is to repeatedly receive a token from the left (3), and send it to the right (4). The programmer organizes the application such that periodically liveness checking is performed (5). At that point, compromised threads are detected and recreated as long as there are available resources.

```
thread_fn (left, right) {
  if(my group_id ==1)                  // 1
     group_send(right, token)          // 2
  while(true) {
     token = group_recv (left)         // 3
     group_send (right, token)         // 4
     if (time_expired)
       liveness_check ()               // 5
  }
}
```

Figure 7. Abstract Code for Threads in Figure 2

Figure 8 depicts the state of the example application when either processor 3 or its network connection is compromised. As a result of this compromise, two threads, one from group 1 and one from group 2 are compromised.

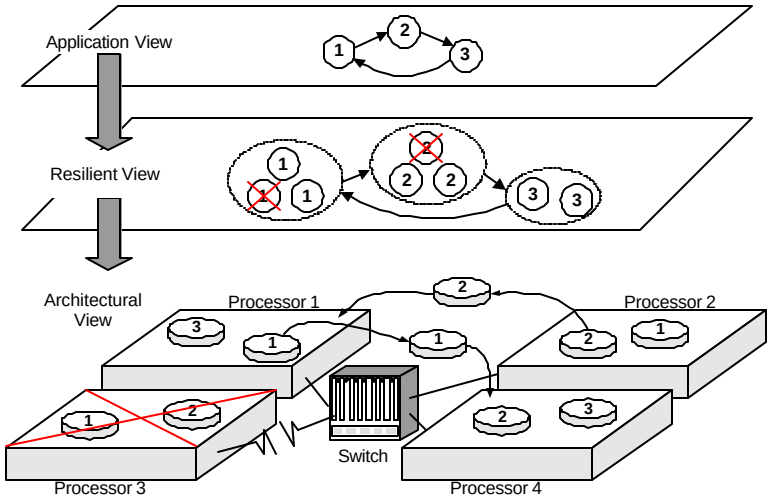


Figure 8. Failure and Reconfiguration

At the next liveness these threads are recreated at processor 1 and 4 respectively as shown in Figure 9. The new threads have the same computation state and communication structure as the residual, uncompromised, threads in their groups. As a result, the required level of resiliency is re-established and the application can tolerate further attacks in the future. Notice that the reconfiguration of the compromised threads is transparent, as there are no changes at the application layer.

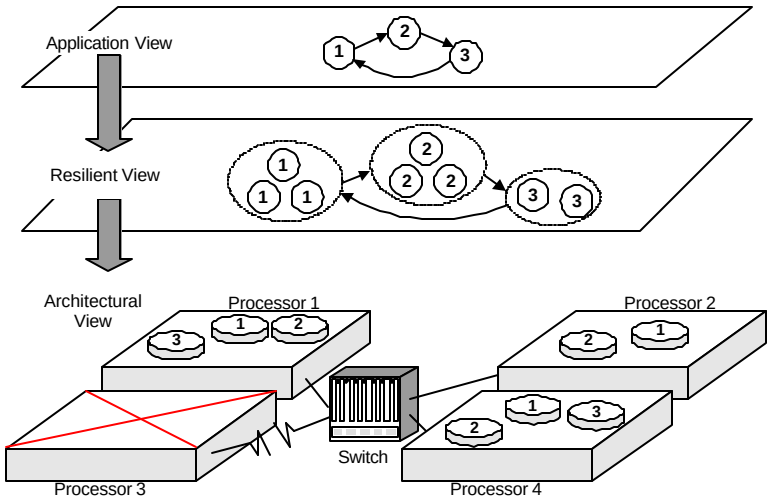
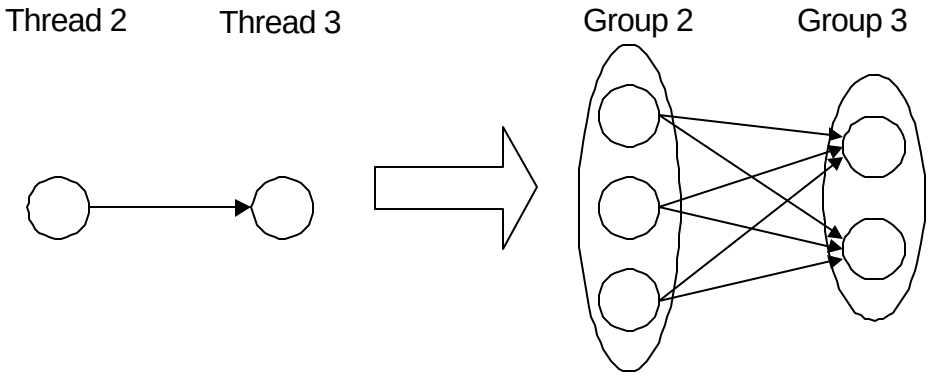


Figure 9. After Liveness Checking

Flow Control Protocol

The flow control protocol ensures reliable, ordered delivery of messages between the groups in the presence of a compromise. Figure 10 shows the impact of replication on the communication structure in Figure 2. At the application layer, threads 2 and 3 communicate directly through a single point-to-point channel. At the resilient layer, the sender has replication level 3 and is represented by group 2, while the receiver has replication level 2 and is represented by group 3. The actual communication structure implemented to achieve this group communication is shown on the right. Each thread in group 2 replicates its communication to group 3. This communication is hidden from the programmer in that it is provided by virtue of the implementation of group communication.



(a) Application View (b) Resilient View

Figure 10. Group Channel Implementation

Figure 11 shows how the flow control protocol effects message transport. Notice that the sending group on the left has three members (resiliency of level 3) and the right hand side shows one of the receiving threads. The receiving thread may receive the same message

three times without error or less than three due to compromises. In this picture, the third channel has failed and no more messages are transmitted after the fourth message over this failed channel. The receiver discards the duplicated messages, reorders the incoming messages, and delivers them to the application level thread. The shaded messages in the picture are duplicates that are received but discarded.

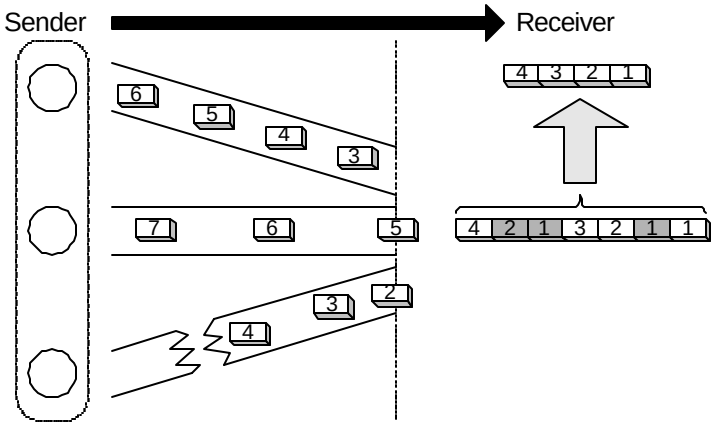
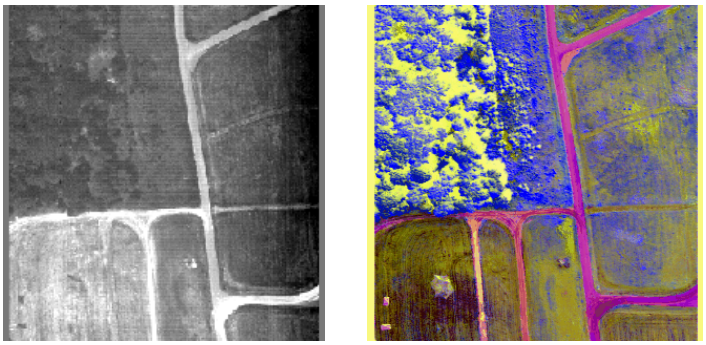


Figure 11. Flow Control

CONCURRENT REMOTE SENSING

The first application to which we have applied resiliency is a concurrent *spectral-screening Principal Component Transform algorithm* (s-PCT) that can be used for remote sensing applications [27]. The algorithm takes as input a large number of grey-scale images emanating from a hyper-spectral sensor. Each image corresponds to a particular wavelength of light, for example, Figure 12a shows the image taken at 1998nm using a 210-channel hyper-spectral image collected with the Hyper-spectral Digital Imagery Collection Experiment (HYDICE) sensor, an airborne imaging spectrometer. The HYDICE image set corresponds to foliated scenes taken from an altitude of 2000 to

7500 meters at wavelengths between 400nm and 2.5 micron. The scenes contain mechanized vehicles sitting in open fields as well as under camouflage. The s-PCT algorithm removes redundancy in the image set and presents a single color composite image that shows the important spectral contrast. For example, Figure 12b shows the output of the algorithm in which the mechanized vehicles are clearly visible in the lower left of the figure due to spectral contrast.



(a) (b)

Figure 12. Concurrent Remote Sensing

The distributed version of the s-PCT algorithm uses the standard manager/worker decomposition technique [24] as shown in Figure 13. A sensor thread generates and partitions the 210-frame image cube into sub-cubes and distributes the sub-cubes to worker threads. A manager synchronizes the actions of these workers, accumulates partial results, and displays the resulting image.

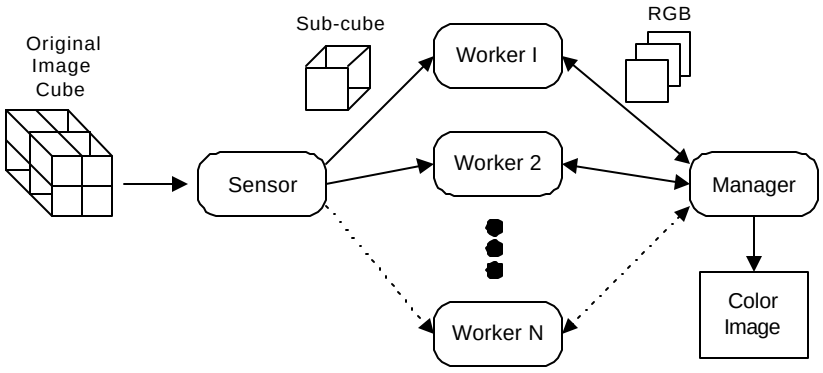


Figure 13. Manager/Worker Communication Model

Each worker thread executes the algorithm shown in Figure 14 and maintains a set of sub-cubes (lines 1, 4) to operate on. An initial request is sent to the sensor to obtain the first sub-cube (line 2). After this initial request, the processing of each sub-cube is overlapped with communication of the next remaining sub-cube from the sensor (line 3). This represents the primary communication step in the algorithm and corresponds to distributing $1/n^{th}$ of the image cube to each of n worker threads.

The spectral screening algorithm produces a set of unique spectra. Although each sub-cube contributes to this set through an appropriate abstract operation (line 6), the set must be accumulated across all sub-cubes. This accumulation is performed through communication with the manager. Each worker sends a prospective subset of the spectra to the manager (line 7) and overlaps this communication with computation of the next subset. When all sub-cubes have been processed, the manger transmits the resulting unique set to all workers (line 8). Typically, the amount of communication in this step is orders of magnitude less than the size of an image cube.

When the spectral screening is completed globally, the algorithm proceeds to compute a set of statistics (mean-vector and covariant-sum) that give a measure of the variation in

images at each spectra. Although, once again, the statistics can be largely computed on a per sub-cube basis using an appropriate abstract operation (line 9), the manager is again involved in assembling the statistics to form a transformation matrix A and mean-vector m (lines 10, 11). The communication involved in this step is on the order of n^2 where n is the number of spectra, again typically significantly smaller than the size of the image cube.

With the matrix A and mean-vector m available, a PCT (line 12) and human-centered mapping (line 13) can be computed on each sub-cube independently to produce a patch of the final color image. The patches are accumulated at the manager for display (line 14). Thus, the final communication is only m^2, where m is the size of the image. Periodic liveness checking is performed when appropriate (line 15).

```
worker_fn() {
   cubes = {}                                                    // 1
   group_send(request,sensor)                                    // 2
   while(numsubcubes <= numcubes/numworkers) {
         subcube = recv(sensor)                                  // 3
         cubes = cubes U subcube                                 // 4
         group_send(request,sensor)                              // 5
         ssubset = spectral_screening(subcube)                   // 6
         group_send(ssubset, manager)                            // 7
   }
   sset = group_recv(manager)                                    // 8
   substats = statistics(sset)                                   // 9
   group_send(manager, substats)                                 // 10
   [A, m] = group_recv(manager)                                  // 11
   subcomponents = PCT(A, m, cubes)                              // 12
   subimage = human_centered_mapping(subcomponents)              // 13
   group_send(subimage, manager)                                 // 14
   if (time expires)
     liveness_check ()                                           // 15
  }
}
```

Figure 14. Abstract Code for Worker Thread

In our experiments, the threads in this program were partitioned to execute on up to 64 processors. The performance of the algorithm was measured on the distributed environment that was organized as 32 Pentium 400 MHz dual-processor computers running the Linux operating system, and connected with 100BaseT networking technology. The sensor and manager were mapped to one machine, while each of the remaining machines executed workers. Resiliency was applied uniformly to harden the application by replicating the worker elements. The manager and sensor were not replicated since the sensors are hardware components and the manager is mapped to a single display device. Figures 15, 16, and 17 show representative experimental results from a broad set of experiments that we have conducted to measure the overhead caused by resiliency and liveness checking. The application was executed once for Figures 15 and 16, and 100 iterations for Figure 17. Three parameters were varied in the experiments: the number of processors (1 to 64), the level of replication (1, 2, 3, or 7), and the frequency of the liveness checking (0 to 100 checks over the course of the 100 iterations).

Although resiliency rather than scalability of the concurrent algorithms is the subject of this paper, it is valuable to ensure that the use of resiliency does not dramatically impact scaling properties. Figure 15 shows the scalability of the concurrent algorithm by measuring the execution time of a single remote sensing operation with respect to the varying number of processors and differing levels of replication. Ideal speedup is the maximum speedup obtainable, which is represented by the sequential execution time divided by the number of processors. Uniform decomposition was used, i.e. the number of partitions is equal to that of processors. As with all applications, eventually the effects

of diminished granularity outweigh the performance improvement associated with concurrency; each component of work is reduced to the point where the cost of communication dominates, for example, the ratio of communication to total execution time increases from 4% with 8 processors to 54% with 64 processors. From 2 processors to 4 processors, the performance improved 97% while from 32 processors to 64 processors only 38% improvement was observed.

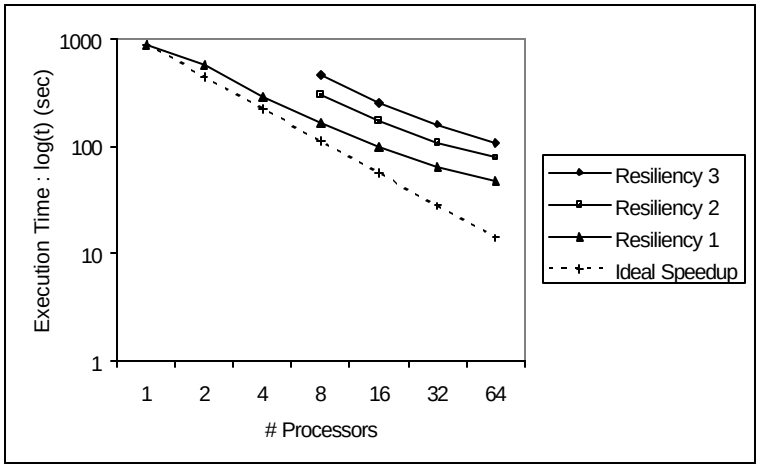


Figure 15. Scalability of Concurrent PCT

Figure 16 shows the overhead of resiliency with respect to the number of processors (8, 16, 32, and 64). Our expectation was that since replication of a thread doubles its computational requirements, level 2 and level 3 resiliency would execute with a two or three-fold decrease in speed respectively. The results indicate however, that in fact performance did not decrease linearly with the level of replication and was less than expected for all the decompositions. Notice that the overhead caused by resiliency 2 is only 82% over resiliency 1, and 186% for resiliency 3 respectively. This artifact resulted from the overlapping of communication and computation in the resilient application: Idle

time in the application allowed cycles to be used in completing replicated tasks that would have otherwise been wasted. Obviously, this phenomenon is highly application dependent, however, idle cycles can occur for many reasons in distributed applications, e.g. file I/O, synchronization, global operations, etc. Therefore it is not unreasonable to assume that resiliency may often be achievable without significant computational costs.

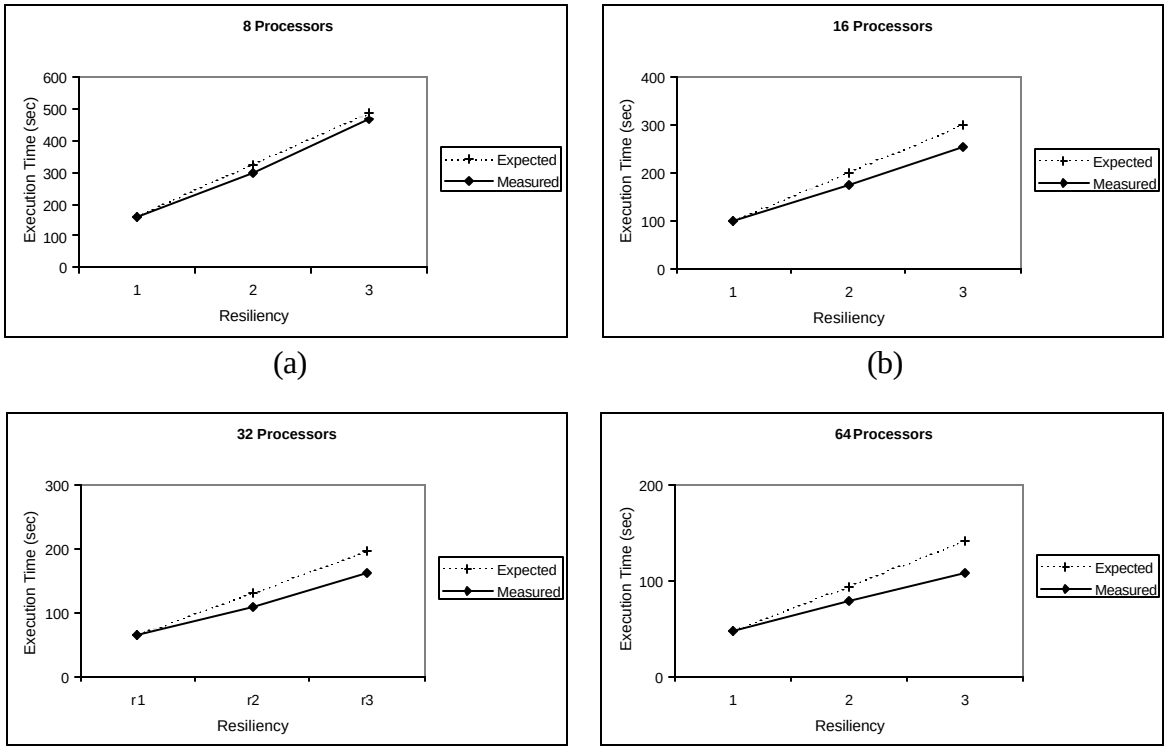


Figure 16. Overhead of Resiliency

Example results concerning the frequency of liveness checking for both synchronous and asynchronous liveness checking mechanisms are presented in Figure 17. For these results, the problem was decomposed into 16. The lower line shows the performance of resiliency 3 using 32 processors. The top line represents resiliency 7. It used more processors, 56, for the same number of decompositions to avoid excessive load of

computation per processor. Even though resiliency 7 may seem to be a high level of replication, we consider this case interesting since it more closely approximates the computational model presented in Section 1. These results show that the use of liveness checks does not incur noticeable overhead for all cases. The overhead was less than 1% for both synchronous and asynchronous liveness checking methods even when liveness checking is frequent (every iteration) and the level of resiliency is high, i.e. 7. This is beneficial in that frequent use of liveness checking allows an application to recover from the possible failure more quickly.

There was no performance difference between synchronous and asynchronous liveness checking methods in this particular application. The reason was that this application has SPMD style parallelism so that the workload was evenly distributed over the threads, and that the experimentation was done on homogeneous systems. Therefore, most of the threads perform the liveness checking almost at the same time. In addition, the overhead of resiliency for level 7 was only 79 % over resiliency 3, indicating that very high levels of survivability may be possible without a direct linear cost.

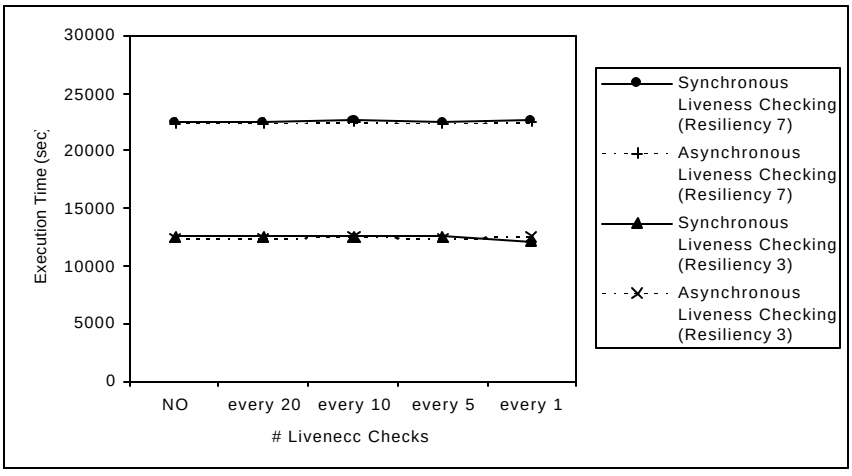


Figure 17. Overhead of Liveness Checking

Dirichlet Boundary Problem

We have developed another technology demonstration program in the fluid dynamics area, the Dirichlet boundary problem. The Dirichlet boundary problem is a numerical simulation problem on a two dimensional grid. Each point on the grid has an (x, y) location and a value representing temperature of some material. At each time step, each point's temperature is averaged with its neighbor's temperatures to find the point's temperature at the end of the time step. This operates for all grid points that are not on the boundary. Boundary grid points are assumed to have a constant value. The Dirichlet problem is simple in that the workload is uniform. This allows the domain decomposition technique to be used in dividing up the workload among processes. A two-dimensional decomposition involves partitioning the grid by both rows and columns (i.e. into patches). In our explanation, two-dimensional decomposition is used. The Dirichlet boundary problem can be parallelized reliably using computational resiliency. Figure 18 describes the abstract code of what each node performs.

```
Entire two dimensional grid is decomposed into the patches
Computing node is created
Set up the communication channels among the nodes
Each node is assigned a patch with initial boundary condition
Initialize each node state
while (not the norm is converged to an acceptable point) {
   for all neighbors {
      send the edges to the neighbor
    }
   for all neighbors {
      receive the edges from the neighbor
   }
   calculate the new temperature at each grid point of the patch
   calculate the new norm
   if (time expires) perform liveness checking
}
```

Figure 18. Dirichlet Boundary Problem and Its Parallelization

Each patch is assigned to a different processor for parallel computation. Figure 19 shows how a two dimensional grid is mapped into a matrix of processors, how each workload is assigned to a node, and how necessary communication paths are set up. In the right picture, each circle represents a node mapped to each processor and the arrows represent the communication paths among the processors.

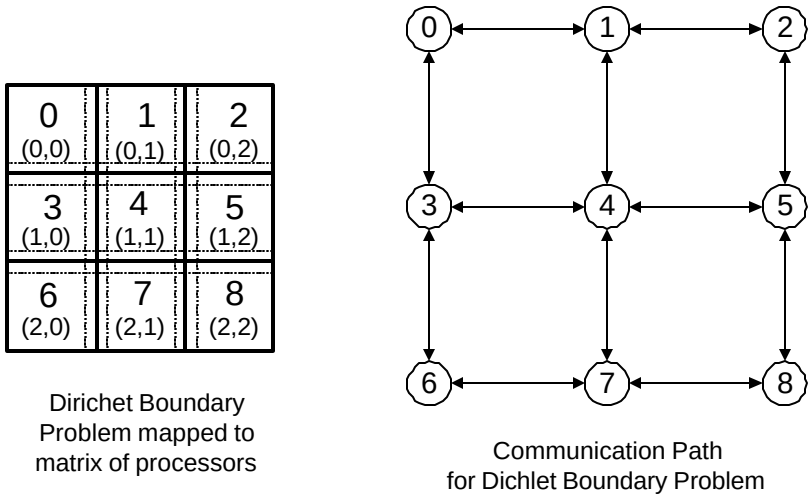


Figure 19. Dirichlet Boundary Problem and Its Parallelization

The performance of the algorithms was measured on the same distributed environment used for concurrent remote sensing in the previous section. The same experiment was conducted with all threads replicated up to the level of seven. Grid size was 6400×6400. Figure 20 shows the speedup gained as a function of the number of processors both with and without resiliency. Due to the memory limitation, the algorithm couldn't run on less than 4 processors. Once again, as we would expect, eventually granularity concerns begin to reduce the speedup of the algorithm. For example, from 4 to 8 processors, the

performance improved 88% while from 32 processors to 64 processors only 60% improvement was achieved.

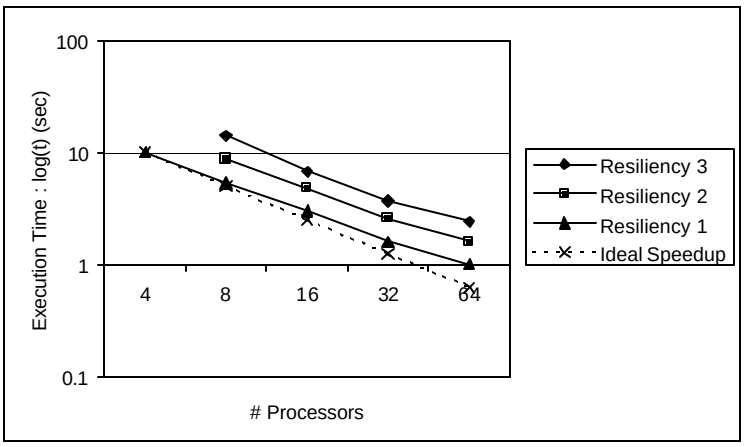


Figure 20. Scalability of Concurrent Dirichlet Boundary Problem

Figure 21 shows the overhead of resiliency with respect to the number of processors. Once again, when resiliency was applied the expected result was that performance would decrease by a factor of two or three depending on the specified resiliency since the replicated processes require both memory and processor resources. The overhead caused by resiliency 2 was 64% over resiliency 1 and 166% for resiliency 3 respectively. Higher improvement rate was achieved compared to the previous application, which resulted from the fact that Dirichlet boundary problem has a lower communication to computation ratio. Communication overhead was the main performance bottleneck with a higher number of processors for the previous application. However, the Dirichlet boundary application has less communication overhead even with higher number of processors. For example, the ratio of communication to total execution time was 20% compared to 54% for the concurrent remote sensing application with 64 processors.

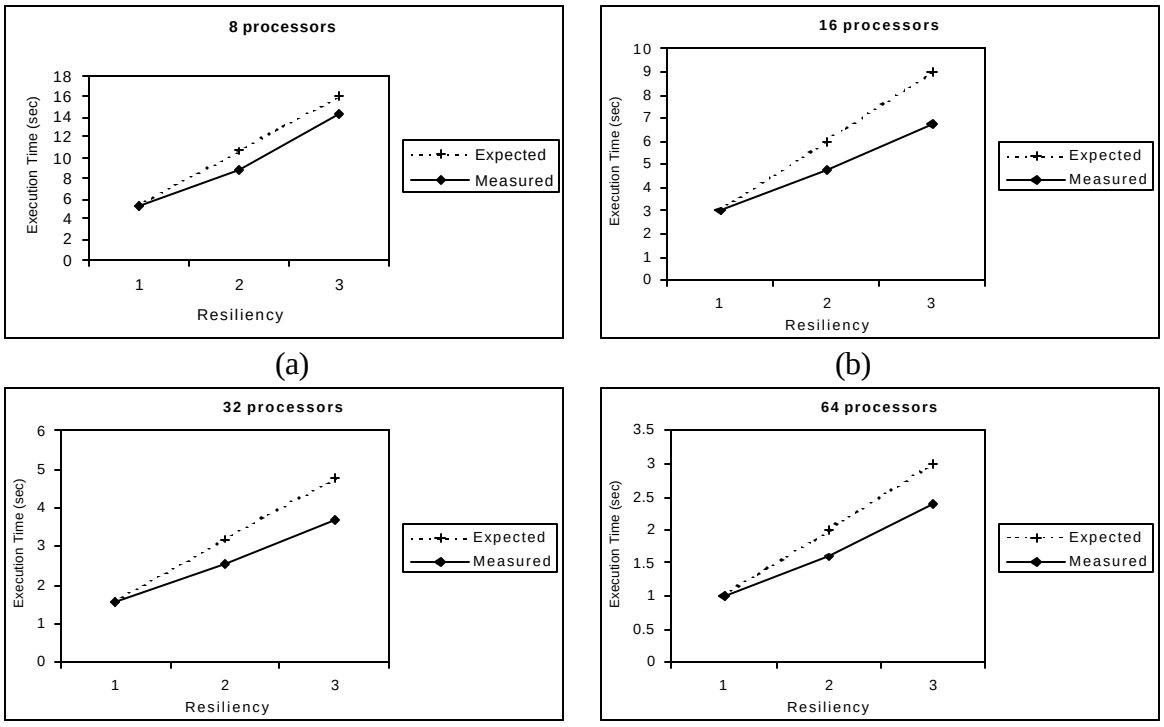


Figure 21. Overhead of Resiliency

Figure 22 shows the overhead of liveness checking for two different liveness checking methods. Unlike the previous application, the results indicate that the asynchronous liveness checking yields a slightly better performance, especially with a higher frequency of liveness checking and resiliency, which was 5% improvement for resiliency 7 and 100 liveness checks. Processors in the middle of the grid need to exchange the boundary information with four neighboring processors while the processors around the boundary with two or three neighbors. Therefore, the processors have different communication overhead, which results in a difference in execution time. With asynchronous liveness checking, the faster groups don't have to wait for the slow groups for liveness checking.

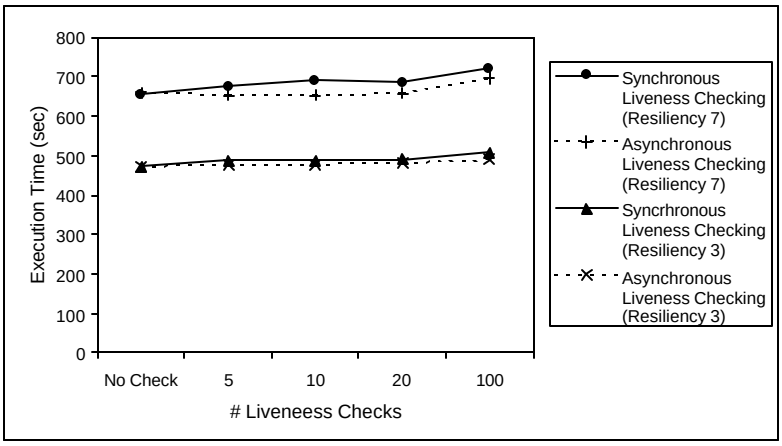


Figure 22. Overhead of Liveness Checking

CONCLUSION

This paper has described the notion of computational resiliency and discussed the implementation issues associated with a prototype-programming library that supports the idea. The paper shows how the concepts and library can be applied in the context of a realistic military application, a concurrent remote sensing, and a parallel application, Dirichlet boundary problem. The implementations of these applications were studied to ascertain the overheads associated with the technology on a moderately scaled, homogeneous architecture consisting of 32 high-performance dual-processor PC's connected with 100Mbit/sec Ethernet technology.

For both applications, ability to utilize idle cycles to reduce the cost of increased survivability was evident, especially at higher levels of redundancy than one normally considers practical. This higher level is directly motivated by the computational model which provides strength in numbers. Although initially, the use of group based liveness checking was considered to be a significant defect with the current implementation strategy, it has proved to be less problematic than expected accounting for less than a 1%

overhead in both applications. We have developed two different liveness checking methods to support a wider range of applications and showed how they can affect the performance of the applications.

Many aspects of computational resiliency remain to be explored and several alternative implementation strategies have yet to be tested. However, the results in this paper indicate that the general concept is both practical and has less cost than originally anticipated. We are currently exploring the related issues of resource management, thread camouflage, and a passive replication based approach.

REFERENCES

1. Lee J, Taylor S. Advances in computational resiliency. *IEEE Aerospace Conference,*. Big Sky, Montana, March 2001.
2. Achalakul T, Lee J, Taylor S. Resilient image fusion. *IEEE ICPP Workshop on High Performance Scientific and Engineering Computing with Applications (HPSECA)*, Toronto, Canada, August 2000.
3. Guerraoui R, Schiper A. Software-based replication for fault tolerance. *IEEE Computer* 1997; 68–74.
4. Budhiraja N, Marzullo K, Schneider FB, Toueg S. Primary-backup approach. *Proceedings of the Sixth International Workshop on Distributed Algorithms*, Haifa, Israel, 1992.
5. Schneider FB. Implementing fault-tolerant services using the state machine approach : a tutorial. *ACM Computing Surveys* 1990; **22**(4):299–319.
6. Sussman JB, Marzullo K. Comparing primary-backup and state machines for crash failures. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, 1996.
7. Cheriton DR, Zwaenpoel W. Distributed process groups in the V-Kernel. *ACM Transactions on Computer Systems* 1985; **3**(2):77–107.
8. Birman KP, van Renesse R. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press: Los Alamitos, CA, 1994.
9. van Renesse R, Birman KP, Maffeis S. Horus: A flexible group communication system. *Commumunications of ACM* 1996; **39**(4).
10. Amir Y, Dolev, Kramer S, Malki D. Transis : A communication sub-system for high availability. *Proceedings of the 22nd Annual Internaltional Symposium on Fault-Tolerant Computing*, July, 1992; 76–84.
11. Agarwal DA. Totem: A reliable ordered delivery protocol for interconnected local-area networks. *PhD Thesis*, Department of Electrical and Computer Engineering. University of California, Santa Barbara, 1994.
12. Kaashoek MF, Tanenbaum AS, Verstoep K. Group communication in amoeba and its applications. *Distributed Systems Engineering* 1993; **1**(1):48–58.

13. Sunderam VS. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience* 1990; 2(4):315–339.
14. Gropp W, Lusk E, Skjellum A. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MPI Press, 1995.
15. Jajodia S, McCollum CD, Ammann P. Trusted recovery. *Communications of ACM* 1999; 42(7).
16. Plank JS, Beck M, Kingsley G, Li K. Libckpt: Transparent checkpointing under Unix. *USENIX Winter 1995 Technical Conference*, 1995.
17. Johnson DB. Distributed system fault tolerance using message logging and checkpointing. *PhD Thesis*, Rice University, 1989.
18. Ramkumar B, Strumpen V. Portable Checkpointing for heterogeneous architectures. *27th International Symposium on Fault-Tolerant Computing*, Seattle, Washington, June, 1997; 58–67.
19. Scales DJ, Lam MS. Transparent fault tolerance for parallel applications on networks of workstations. *Proceedings of the 1996 USENIX Technical Conference*, January, 1996.
20. Taylor S, Watts J, Rieffel M, Palmer M. The concurrent graph: Basic technology for irregular problems. *IEEE Parallel and Distributed Technology* 1996; 4(2):15–25.
21. Watts J, Taylor S, Nilpanich S. SCPlib: A concurrent programming library for programming heterogeneous networks of computers. *IEEE Information Technology Conference*, EX 228, 1998; 153–156.
22. Watts J, Taylor S. A practical approach to dynamic load balancing. *IEEE Transactions on Parallel and Distributed Systems* 1998; 9:235–248.
23. Koniges AE. *Industrial Strength Parallel Computing*. Morgan Kaufmann, 2000; 147–168, 227–246, 267–296.
24. Seitz CL. The cosmic cube. *Communications of ACM* 1985; 28(1):22–33.
25. Bokhari SH. On the mapping problem. *IEEE Transactions on Computers* 1981; 30(3):207–214.
26. Chandy LM, Taylor S. *An Introduction to Parallel Programming*. Jones and Bartlett publishers: Boston, 1992.
27. Achalakul T, Haaland PD, Taylor S. Mathweb: A concurrent image analysis tool suite for multi-spectral data fusion. *Sensor Fusion: Architectures, Algorithms, and Applications III* (*Proceedings SPIE*, vol. 3719). 1999; 351–358.

Authors' biographies:

Joohan Lee is a Post Doctorate at the Center for Systems Assurance in the Computer Science Department of Syracuse University. He received his BS and MS degrees in Computer Science from Sogang Universitry, Korea in 1993 and 1995, respectively. He holds a PhD in Computer Science from Syracuse University, gained in 2001. His main research interest includes fault-tolerant distributed systems, parallel processing, network security, and Internet technologies.

Steve J. Chapin is an Associate Professor in the Department of Electrical Engineering and Computer Science and Director of the Center for Systems Assurance at Syracuse University. He received a Bachelor of Science in both Mathematics and Computer Science from Heidelberg College in 1985. He pursued his graduate study at Purdue University, earning a Master of Science in Computer Science in 1988 and a PhD in Computer Science in 1993. His research areas are operating systems, distributed systems, and computer and network security.

Steve Taylor is a Professor at Thayer School of Engineering, Dartmouth College. He works for the Institute of Security Technology. His research interests include cyber forensics, network security, information warfare, real-time sensor fusion, distributed computing and Web technologies, high-performance distributed applications, concurrent programming, multi- and hyper-spectral image analysis and program design methods.