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. The notion of provides both software fault tolerance and the ability to tolerate information warefare(IW) attacks. This technology seeks to strengthen a military mission, rather than 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 automatically regenerate replication in response to a failure or attack, allowing the level of system reliability to be restored and maintained. Replicated structures can be protected through several techniques such as camouflage, dispersion, and layered security policy. 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*—Software reliability, computational resiliency, fault tolerance, distributed computing, information warfare, network security, heterogeneous computing, load balancing**

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 may become available dynamically through retasking.

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.

Joohan Lee is with the Computer Science Department, Syracuse University, Syracuse, NY 13244 USA (telephone: 315-498-4434, e-mail: jlee@ ecs.syr.edu).

Steve J. Chapin is with the Computer Science Department, Syracuse University, Syracuse, NY 13244 USA. (e-mail: chapin@ecs.syr.edu).

Stephen Taylor is with Institute of Security Technology Studies, Thayer School of Engineering, Dartmouth College, Hanover, NH 03755 USA (e-mail: steve@ists.dartmouth.edu).

We are investigating an alternative model of distributed computation termed *computational resiliency*. This model combines real-time attack assessment with process reconfiguration, dispersion, camouflage, on-the-fly replication, and layered security policy to reliably maintain information operations. 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)[1]. 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 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 support this model, we have developed an application independent, programming technology that operates in heterogeneous distributed computing environments. The technology can be applied either to an entire application or a small number of 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 homogeneous 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 homogeneous cluster, no byte or machine translations are needed.

Since machines in the environment may have widely different performance and memory characteristics, load balancing techniques are required. These techniques must disperse replicated structures to realize improved reliability. To explore the performance issues associated with these concepts, we have incorporated the technology into two prototype distributed

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

applications: a towed array sonar and a hyper-spectral remote sensor. In this paper we outline the applications, and show how resiliency is applied to them. Performance measurements are provided that quantify the overhead of resiliency, under normal operating conditions, using a network architecture containing 21 heterogeneous 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. Here we are concerned 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 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, and transparently recover system function.

In many client-server style applications, the techniques employed to provide recovery can be divided into two 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 significant degradation in system response. In active replication [3], all replicas have the same level of 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 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 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 [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, 15, 16, 17].

The use of networks of personal computers, workstations, and symmetric multiprocessors as a computing platform requires load balancing techniques. Computers in a typical network often differ in processor performance and memory characteristics. Many load balancing techniques have been developed for heterogeneous environments[18,19]. These typically assume that attacks or faults are unlikely and focus on the optimal allocation of resources. Load balancing techniques for efficient allocation of the replicated processes have also been studied in fault tolerant systems [20,21,22,23]. For example, Nieuwenhuis [20] has proposed a static model to derive the mapping of replicated processes. Bannister [22] has presented an algorithm that balances the load of replicated processes over a homogeneous system and subsequently analyzed the performance of the algorithm. Schatz [23] proposed 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 heterogeneous systems.

The starting point for the work described in this paper is the Scalable Concurrent Programming Library (SCPlib) [24,25,26]. This library provides a heterogeneous 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 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). 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 load balancing and granularity control techniques based on thread migration.

III. Computational Resiliency

To provide reliable operation, applications may 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 in that 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.

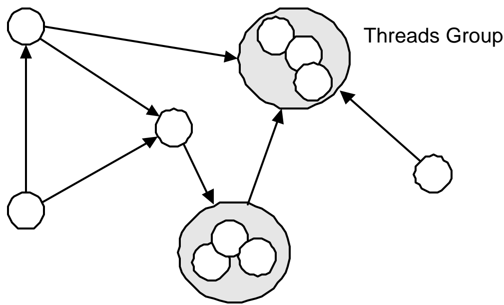


Fig. 1. Replication of Threads

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 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.

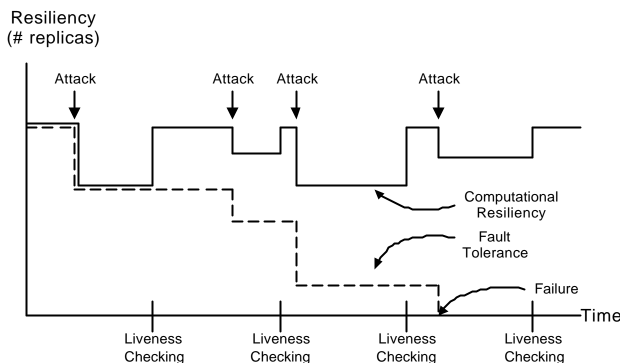


Fig. 2. Fault-Tolerance vs. Computational Resiliency

Using resiliency, periodic liveness checks are performed. These checks 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.

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 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 to appropriate processors such that replicas in a single group are placed in different processors at the architectural level.

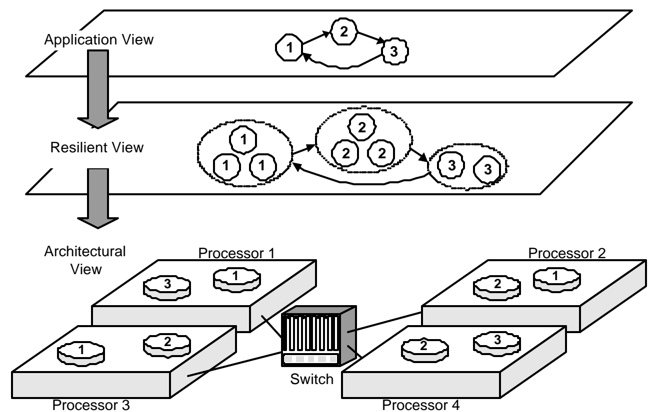


Fig. 3. Computational Resiliency Using a Cluster of Multiprocessors

IV. Load Balancing Algorithm

Traditional bad balancing techniques address the optimal allocation of resources to tasks. We augment this process with reliability constraints. Fig. 4 outlines the greedy algorithm used, where l_j represents the load of task *j*, L_i the load on processor *i*, r_j the number of replicas for task *j*, T_i a set of tasks mapped to computer *i*, and S_j a set of computers to which task *j* is may not be allocated.

To assess the load of a task, we 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. We 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 since 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 may reduce reliability. Thus we ensure that replicas within a group are allocated to different LANs where possible.

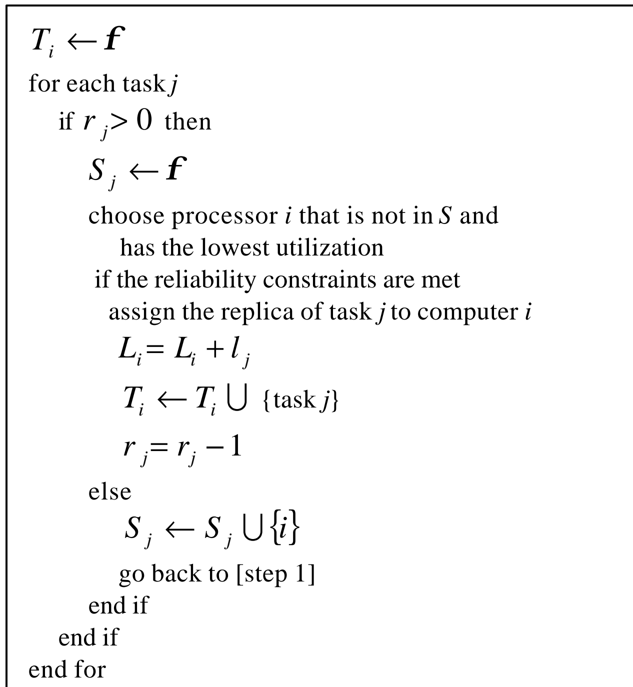


Fig. 4. Load Balancing Algorithm

V. EXPERIMENTAL TESTBED

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

- **Machine 0:** One 4-processor, Pentium III, 450 MHz machine running Windows NT 4.0, with 1.5Gbytes of memory, Gigabit network. (9.3)
- **Machine 1:** One dual processor, Pentium II, 300MHz machine running Windows NT 4.0, with 256Mbytes of memory, GigaBit network. (2.6)
- **Machines 2 and 3:** Two 8-processor Pentium III, 500 MHz machines running Windows NT 4.0, with 4 Gbytes of memory, GigaBit network. (14.4)
- **Machines 4 and 5:** Two Pentium III, 500 MHz machines running Windows NT 4.0, with 128 Mbytes of memory, GigaBit network. (2.6)
- **Machines 6 to 9:** Four Celeron 533 MHz machines running Windows NT 4.0, with 128 Mbytes of memory, 100 BT network. (2.2)
- **Machines 10:** One SGI Indigo II, 200 MHz R4400, running IRIX 6.4, with 128 Mbytes of memory, 100 BT network. (1.0)
- **Machines 11:** One SGI Indigo II, 150MHz R4400, running IRIX 6.4, with 288 Mbytes of memory, 100 BT network. (1.3)
- **Machines 12 to 20:** Nine dual Processor, Pentium II, 400MHz 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 brackets. Note that the performance of the machines varied by a factor of almost 14.4(x8), 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 two separate sub-networks that were connected through the Gigabit Ethernet networking.

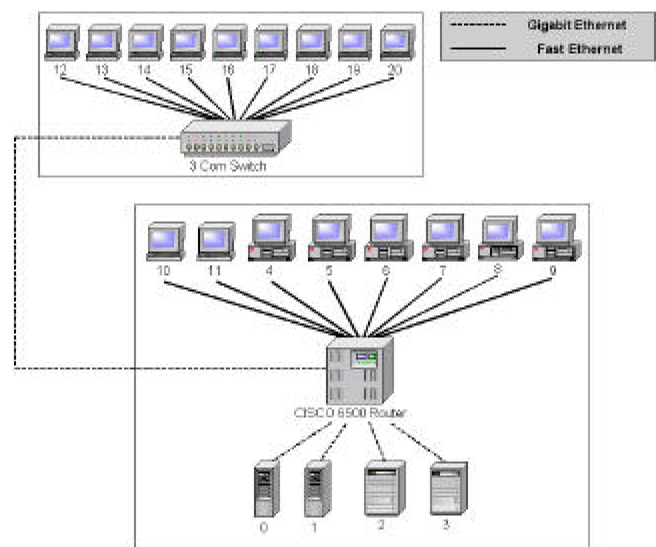


Fig. 5. Heterogeneous Network Architecture

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 beamforming. This process combines the outputs from a number of omni-directional transducer elements, arranged in an array of arbitrary geometry, so as to enhance signals from some defined spatial locations. It also suppresses signals from other non-target obstacles. Beamformers 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, we have developed a concurrent towed array sonar application based on conventional beamforming techniques [30]. The general concurrent structure of this application is shown in Fig. 6 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 *beamformer* threads. Each thread fifo-buffers NS partial returns and repeatedly computes a covariance matrix and a partial beamforming 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.

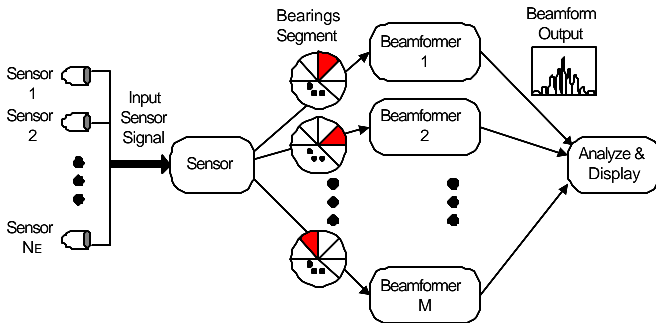


Fig. 6. Communication Model for Sonar Processing

Fig. 7 shows the resilient view of the application where the beamformer threads are replicated with degree two.

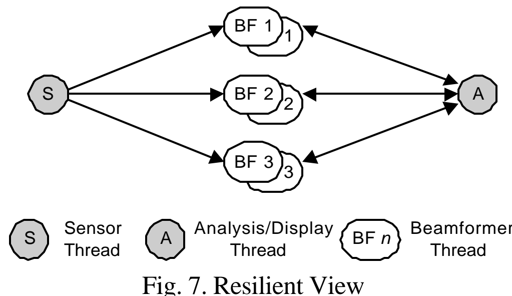
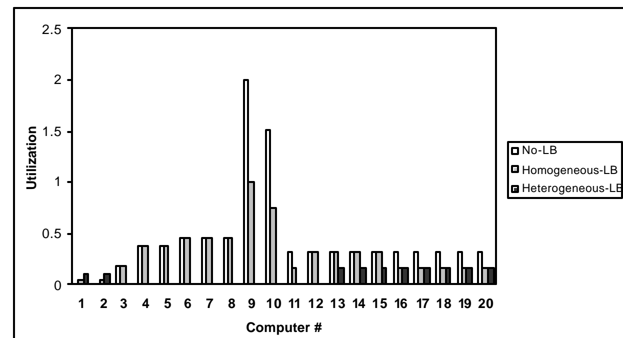


Fig. 7. Resilient View

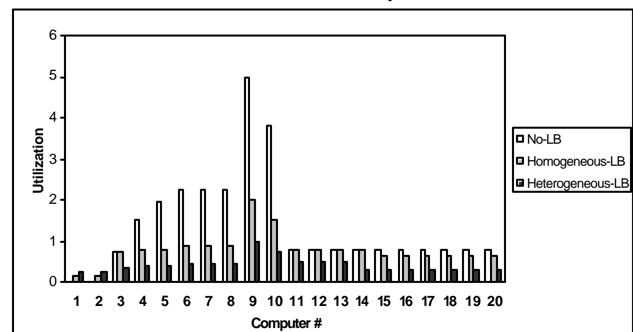
The sensor and display were mapped to Machine 0 in the testbed due to memory concerns, while each of the remaining 20 machines executed beamformers. Resiliency was applied uniformly to harden the application by replicating the beamforming elements. Fig. 8, 9, 10 and 11 show representative experimental results from a broad set of experiments that we have conducted to measure the effectiveness of load balancing and the overhead caused by resiliency and liveness checking. The beamformer was executed once for Fig. 8, 9 and 10, and 100 iterations for Fig. 11. Each iteration processed a single set of buffered returns. Three parameters were varied in the experiments: the load balancing method, the level of replication (1, 3, or 7), and the frequency of the liveness checking (0 to 20 checks over the course of the 100 iterations). Even though resiliency 7 may seem to be a high level of replication, we consider this case interesting to investigate since it more closely approximates the computational model presented in Section 1. The number of sonar elements and the number of buffered returns were fixed to 382 and 1000 respectively.

Three experiments were conducted to evaluate the effectiveness of the different load balancing techniques. First, the problem was run without load balancing (No-LB). Next, load balancing strategy based on the number of processors in each machine was used (Homogeneous-LB). In the third case, a small benchmark problem, roughly 20% as large as the full problem, was run on each machine to assess their relative performance. Using static capacity estimates, the problem was then balanced (Heterogeneous-LB).

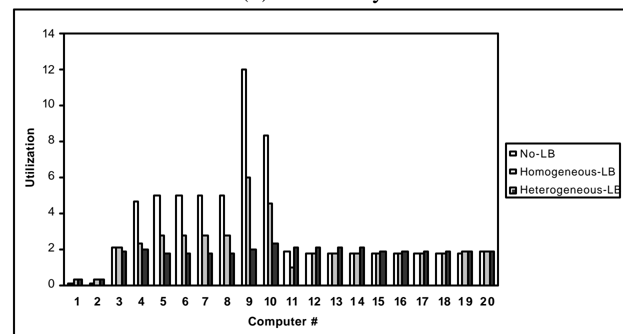
Fig. 8 shows the relative utilization of the computers based on their relative capacity and workload assigned. For resiliency 1, it is permitted that no task is assigned to a slow processor, while in resiliency 3 at least one task is allocated to every computer to ensure higher reliability. For example, in Fig. 8(a), machines 3 to 12 were dropped for resiliency 1. Heterogeneous-LB technique shows the most balanced utilization. Machines 1 and 2 have room to take more tasks but cannot due to the reliability constraints.



(a) Resiliency 1



(b) Resiliency 3



(c) Resiliency 7

Fig. 8. Utilization for Each Load Balancing Techniques

Fig. 9 summarizes the results of theses experiments. With homogeneous-LB, a 1.9 fold performance improvement was observed with resiliency 7. With heterogeneous-LB, a 2.7 fold performance improvement was observed with resiliency 7.

Scenario	Step Time (sec)	Improvement
No LB	278.9	N/A
Homogeneous-LB	146.9	1.9x
Heterogeneous-LB	103	2.71x

(a) Resiliency 7

Scenario	Step Time (sec)	Improvement
No LB	88.4	N/A
Homogeneous-LB	50.0	1.77x
Heterogeneous-LB	35.4	2.5x

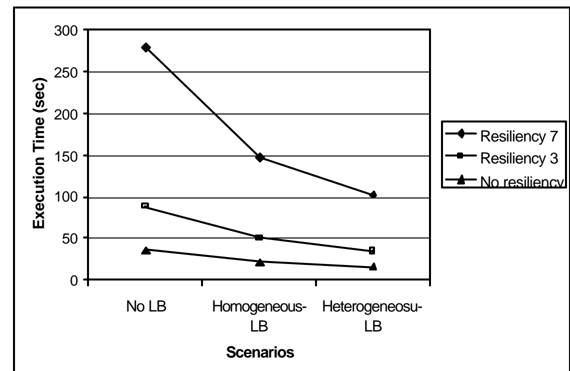
(b) Resiliency 3

Scenario	Step Time (sec)	Improvement
No LB	36.2	N/A
Homogeneous-LB	22.0	1.65x
Heterogeneosu-LB	16.1	2.25x

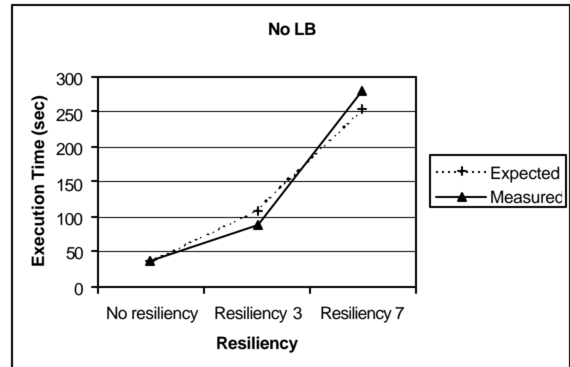
(c) No Resiliency

Fig. 9. Results of Load Balancing Experiments for Entire Heterogeneous Testbed

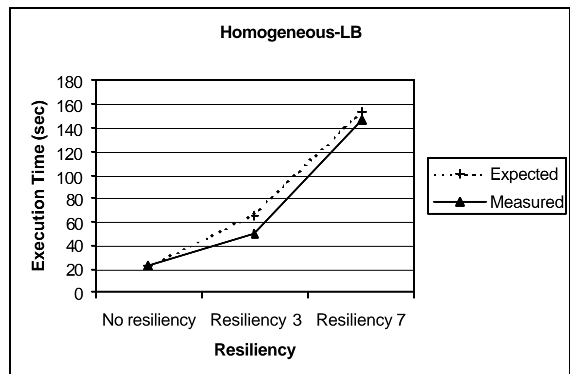
Fig. 10 shows the overhead of resiliency with respect to each load balancing techniques. Our expectation was that since replication of a thread doubles its computational requirements, level 3 and 7 resiliency would execute with a three and seven-fold decrease in speed respectively. Without any load balancing, Fig. 10(b), execution time for resiliency 7 increased more than a factor of 7. With load balancing, however, the results indicate that performance did not decrease linearly with the level of replication and was less than expected for all the cases. The execution time of resiliency 3 increased only 127% and 120% over resiliency 1 for Fig. 10(b) and (c) respectively. For resiliency 7, it was as much as 568% indicating that very high levels of survivability may be possible without a direct linear cost. This artifact results from the overlapping of communication and computation in the resilient application: Idle time 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.



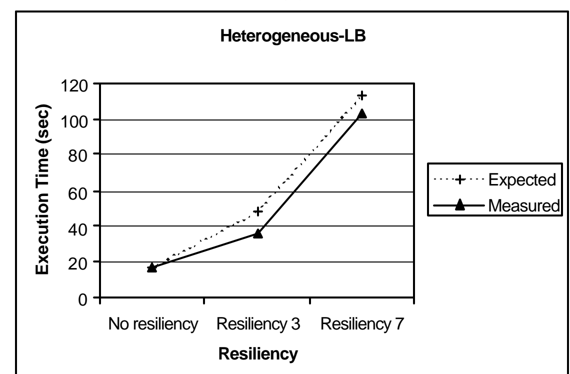
(a)



(b)



(c)



(d)

Fig. 10. Overhead of Resiliency

Fig. 11 shows the cost of liveness checking in this application. The overheads never exceeded 1% even when liveness checking is frequent (once every 5 iterations of the beamformer) and the

level of resiliency is high, i.e. 7.

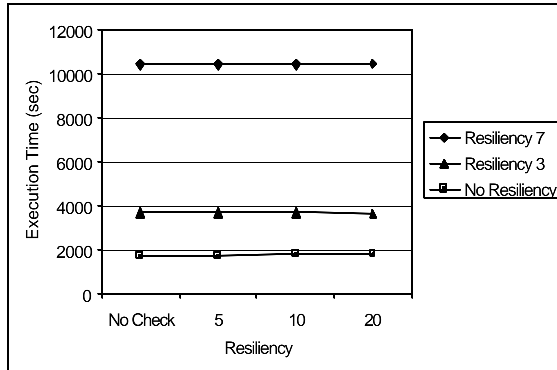
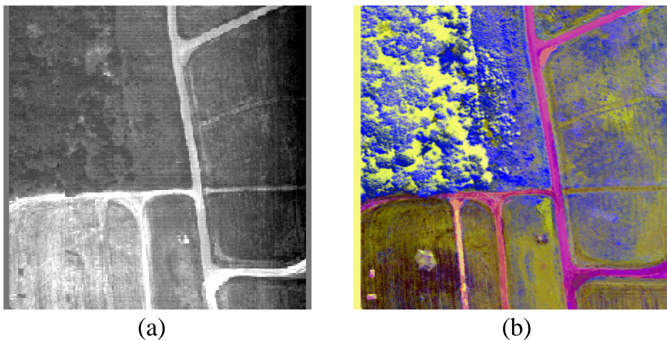


Fig. 11. Overhead of Liveness Checking

VII. CONCURRENT REMOTE SENSING

A second application to which we have applied resiliency is a concurrent *spectral-screening PCT algorithm* (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, for example, Fig. 12(a) 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, Fig. 12(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.



(a) (b)

Fig. 12. Concurrent Remote Sensing

The distributed version of the s-PCT algorithm uses the standard manager/worker decomposition technique [32] as shown in Fig. 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.

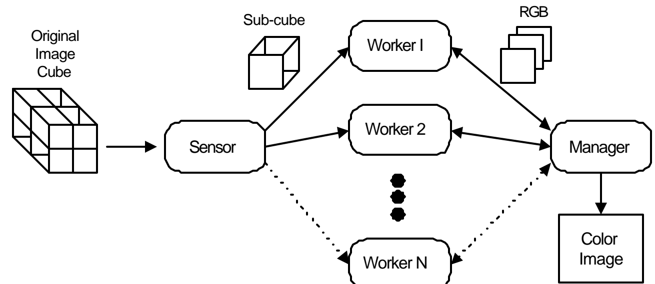


Fig. 13. Manager/Worker Communication Model

Fig. 14 shows the resilient view of the application where worker threads are replicated with degree of three.

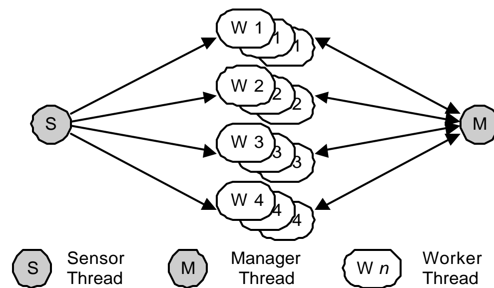


Fig. 14. Resilient View

The performance of the algorithm was measured on the heterogeneous testbed environment. The same experiment was conducted with all workers replicated up to the level of seven; the manager and sensor were not replicated. Fig. 15 shows the results of load balancing experiments; theses 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, i.e. 7.

Scenario	Step Time (sec)	Improvement
No LB	896	N/A
Homogeneous-LB	492	1.82x
Heterogeneous-LB	266	3.37 x

(a) Resiliency 7

Scenario	Step Time (sec)	Improvement
No LB	357	N/A
Homogeneous-LB	197	1.81x
Heterogeneous-LB	158	2.26x

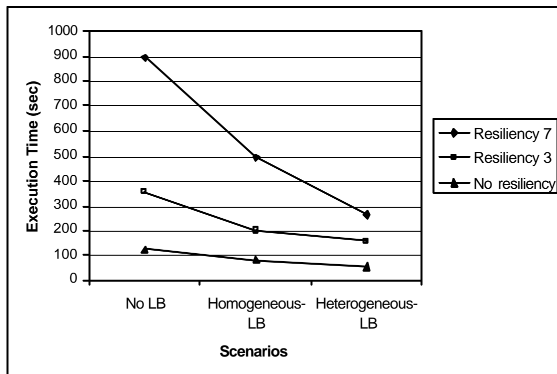
(b) Resiliency 3

Scenario	Step Time (sec)	Improvement
No LB	122	N/A
Homogeneous-LB	81	1.5x
Heterogeneous-LB	54	2.26x

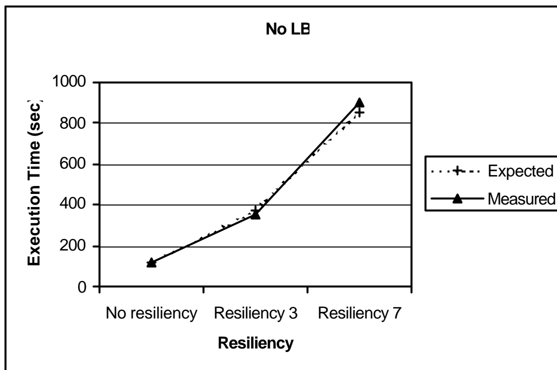
(c) No Resiliency

Fig. 15. Results of load balancing experiments for entire heterogeneous testbed

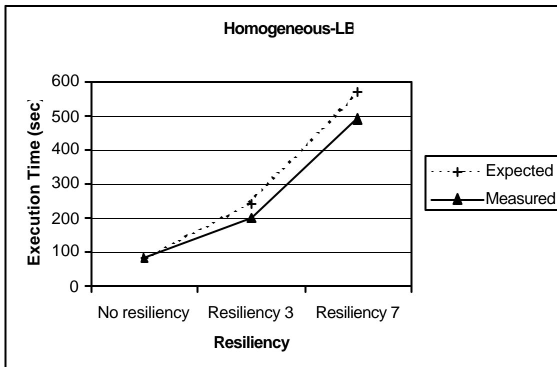
Once again, when resiliency was applied the expected result was that performance would decrease by a factor of three or seven depending on the specified resiliency since the replicated processes require both memory and processor resources. Fig. 16 shows the overhead of resiliency with respect to three load balancing techniques. Without any load balancing, the execution times for resiliency 7 increased more than a factor of 7 in Fig 16(b). Heterogeneous load balancing reduced the overhead of resiliency significantly. With resiliency 7, the overhead was only 393% over resiliency 1 in Fig 16(d). As in the sonar application, we observe that load balancing improved the performance and resiliency is able to utilize idle cycles in the concurrent algorithm to reduce the cost of replication.



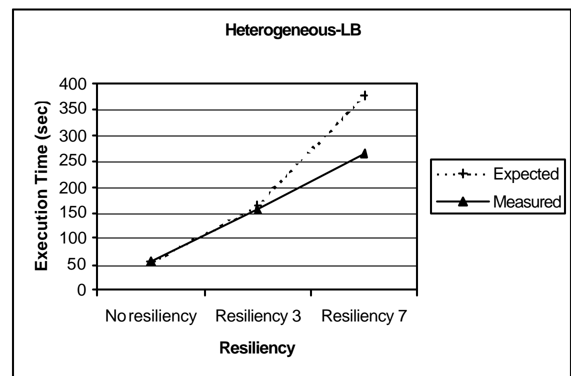
(a)



(b)



(c)



(d)

Fig. 16. Overhead of Resiliency

Fig. 17 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.

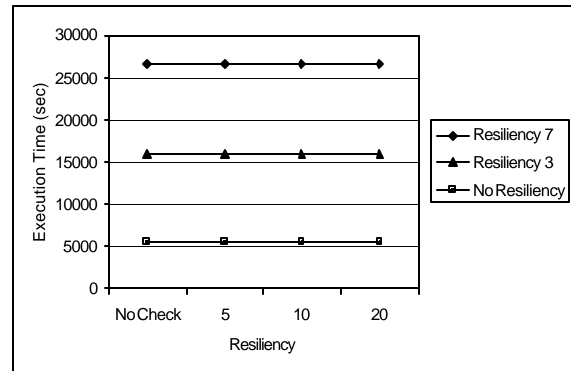


Fig. 17. Overhead of Liveness Checking

VIII. 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 two 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, heterogeneous architecture consisting of computers with varying computing capability, memory availability, operating systems, and networking technology.

For both applications, use of load balancing techniques improved the performance by efficient allocation of the replicated threads. Reliability was considered in the load balancing algorithm to improve the allocation of replicas. The ability to utilize idle cycles significantly 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 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.

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.

ACKNOWLEDGMENT

The authors would like to thank Mike Orlovsky and Thomas Barnard at the Ocean, Radar, and Sensor Systems Division of Lockheed Martin, Gregg Irvin and James Gaska at Mobium Enterprises, and Tiranee Ackalakul and Jeremy Monin at Syracuse University for their aid in constructing the concurrent applications described in this paper.

REFERENCES

[1] Rachid Guerraoui, Andre Schiper, "Software-Based Replication for Fault Tolerance", *IEEE Computer, pp68-74*, April 1997.

[2] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, Sam Toueg, "Primary-Backup Approach", *Proceedings of the Sixth International Workshop on Distributed Algorithms*, Haifa, Israel, 1992.

[3] Fred B. Schneider, "Implementing fault-tolerant services using the state machine approach : a tutorial", *ACM Computing Surveys, Vol 22, No. 4, pp 299-319*, April 1990.

[4] Jeremy B. Sussman, Keith Marzullo, *"Comparing Primary-Backup and State Machines for Crash Failures"*, Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, 1996.

[5] David R. Cheriton, Willy Zwaenpoel, "*Distributed Process Groups in the V-Kernel"*, ACM Transactions on Computer Systems, Vol 3, No 2, pp77-107, Feb. 1985.

[6] Birman, K.P., van Renesse, R., "Reliable Distributed Computing with the ISIS Toolkit", *IEEE Computer Society Press*, Lod Alamitos, Calif., 1994.

[7] van Renesse, R., Birman, K. P., and Maffeis, S., "Horus: A flexible group communication system", *Commumunications of ACM 39, 4*, Apr. 1996.

[8] Amir, Y., Dolev., Kramer, S., and Malki, D., "Transis : A communication sub-system for high availability", *Proc. of the 22nd Annual Internaltional Symposium on Fault-Tolerant Computing, pp 76-84*, July, 1992.

[9] Agarwal, D.A., "Totem : A reliable ordered delivery protocol for interconnected local-area networks", *Ph.d dissertation*, Dept. of Electrical and Computer Engineering. University of California, Santa Barbara, 1994.

[10] Kaashoek, M. F., A. S. Tanenbaum, K. Verstoep, "Group Communication in Amoeba and its Applications", *Distributed Systems Engineering, vol. 1, no. 1, pp 48-58*, September 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] Shushil Jajodia, Catherine D. McCollum, Paul Ammann, "Trusted Recvoery", *Communications of ACM 42, 7*, July 1999.

[14] James S. Plank, Miach Beck, Gerry Kingsley, Kai Li, "Lipckpt : Transparent Checkpointing under Unix", *USENIX Winter 1995 Technical Conference*, 1995.

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

[16] Balkrishna Ramkumar, Volker Strumpen, "Portable Checkpointing for Heterogeneous Architectures", *27th International Symposium on Fault-Tolerant Computing, pp 58-67*, Seattle, Washington, June, 1997.

[17] D. J. Scales and M. S. Lam, "Transparent Fault Tolerance for Parallel Applications on Networks of Workstations", *Proceedings of the 1996 USENIX Technical Conference*, January, 1996.

[18] K. Li, J. Dorband, "A Task Scheduling Algorithm for Heterogeneous Processing", *Proc. High Performance Computing, pp. 183-188*, 1997.

[19] J. Watts, M. Rieffel, S. Taylor, "Dynamic Management of Heterogeneous Resources", *High Performance Computing: Grand Challenges in Computer Simulation, April, pp.151-156*, 1998.

[20] L.J.M. Nieuwenhuis, "Static Allocation of Process Replicas in Fault-Tolerant Computing Systems", *Proc. FTCS-20, June, pp. 298-306*, 1990.

[21] J. Kim, H. Lee, S. Lee, "Replicated Process Allocation for Load Distribution in Fault-Tolerant Multicomputers", *IEEE Transactions on Computers, Vol. 46, No. 4, April, pp. 499-505*, 1997.

[22] J.A. Bannister, 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, M. Goto, "Task Allocation for Maximizing Reliability of Distributed Systems", *IEEE Transactions on Computers, Vol. 41, No. 9, pp. 1,156-1,168,* Sept., 1992.

[24] Taylor S., Watts J., Rieffel M., and Palmer M., "The Concurrent Graph: Basic Technology for Irregular Problems", *IEEE Parallel and Distributed Technology, 4(2): pp15-25*, 1996.

[25] Watts J., Taylor S., and Nilpanich S., "SCPlib : A Concurrent Programming Library for Programming Heterogeneous Networks of Computers", *IEEE Information Technology Conference, EX 228, pp 153-6*, 1998.

[26] Watts J., and Taylor S., "A Practical Approach to Dynamic Load Balancing", *IEEE Transactions on Parallel and Distributed Systems, vol 9, pp 235-248*, 1998.

[27] Industrial Strength Parallel Computing, *Morgan Kaufmann, pp 147-168, 227-246, 267-296*, 2000.

[28] Nielsen R., *Sonar Signal Processing*, Artech House, Inc., 1991.

[29] T. E. Curtis and R. J. Ward, "Digital beam forming for sonar systems", *IEE Proc., Vol. 127, Pt. F, No. 4*, August, 1980.

[30] Thomas Barnard, *Radar and Sonar Signal Processing*, Unpublished, 1998.

[31] Achalakul T., Haaland P. D.,Taylor S., "Mathweb: A Concurrent Image Analysis Tool Suite for Multi-spectral Data Fusion", *SPIE vol. 3719 Sensor Fusion: Architectures, Algorithms, and Applications III, pp 351-358*, 1999.

[32] Chandy L. M., Taylor S., An Introduction to Parallel Programming, *Jones and Bartlett publishers*, Boston, 1992.