International Journal of Contemporary Research in Computer Science and Technology (IJCRCST) e-ISSN: 2395-5325
Volume 2, Issue 9 (September '2016)

CHECKPOINT BASED RECOVERY AWARE COMPONENT SYSTEM IN GRID COMPUTING

A.Suganthi,
M.Phil Research Scholar (PT),
Department Of Computer Science,
Sangunthar Arts and Science College,
Triuchengode,Tamilnadu,India.

R.Bharathi,
Head cum Assistant Professor,
Department Of Computer Science (PG),
Sangunthar Arts and Science College,
Triuchengode,Tamilnadu,India.

Abstract: Grids are distributed systems that dynamically coordinate a large number of heterogeneous resources to execute large scale projects involving collaborating teams of scientists, high performance computers, massive data stores, high bandwidth networking, and/or scientific instruments like telescopes, and synchrotrons. Failure in grids is arguably inevitable due to the massive scale and the heterogeneity of grid resources, the distribution of these resources over unreliable networks, the complexity of mechanisms that are needed to integrate such resources into a seamless utility, and the dynamic nature of the grid infrastructure that allows continuous changes to happen. In this thesis, we propose the Recovery-Aware Components (RAC) approach. The RAC approach enables a grid application to tolerate failure reactively and proactively at the level of the smallest and independent execution unit of the application. The approach also combines runtime prediction with a proactive fault tolerance strategy. By managing failure at the smallest execution unit, and combining runtime prediction with a proactive fault tolerance strategy, the RAC approach aims at improving the reliability of the grid application with the least overhead possible.

Keywords: Grid Computing, recovery, fault tolerance, reliability

I.INTRODUCTION

Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed "autonomous" resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality of service requirements. Grid computing, most simply stated, is distributed computing taken to the next evolutionary level. The goal is to create the illusion of a simple yet large and powerful self-managing virtual computer out of a large collection of connected heterogeneous systems sharing various combinations of resources. Grid is a collection of distributed resources connected by a network, possibly at different sites and in different organizations. Those resources may include supercomputers, instruments such as telescopes and microscopes, computer-controlled factory floor tools, mid- level servers, desktop machines, laptop etc [1]. A grid usually connects huge number of computers over the Internet as a complex computational System. Here numerous tasks are distributed to grid nodes in decentralized fashion. It contains five Components [2].Like A portal, A service broker, Task scheduler, A task manager and A group of grid node. The portal acts as a user interface, through which user can log in and use the grid. After having logged into the grid, a user can submit a task. The service broker will check whether or not the grid possesses resources suitable for handling the submitted task. If so, it will further check whether or not the resources are available now. The task scheduler is responsible for scheduling submitted tasks to be served. The task manager finally launches a submitted task. The nodes, as the core of a grid, are prerequisites. Grid nodes can be desktops, workstations, and clusters that belong to different LANs, WANs, or the Internet.

Grid Architecture: In grid computing, many thousands of small-distributed computing networks would be linked over worldwide grids in a Web like system resembling a public utility's power grid [3]. That will let businesses send data transfer, share software more easily and store even more information than today's computer networks. The development of Grid-enabled applications presents a significant challenge, however, because of the high degree of heterogeneity and dynamic behavior in architecture, mechanisms, and performance encountered in Grid environment. The Grid is made up of a number of components from enabling resources to end user applications. A layered architecture of the Grid and its components are shown in the following, Figure 1.

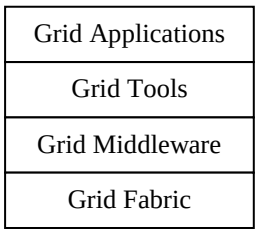


Figure 1: Grid Layer Architecture

Grid computing technologies enable controlled resource sharing in distributed communities and the coordinated use of those shared resources as community members tackle common goals. These technologies include new protocols, services, and APIs for secure resource access, resource management, fault detection, communication, and so forth that in term enable new application concepts such as virtual data, smart instruments. A computational grid can be modeled using 4-layer architecture as (1) Grid fabric, (2) Core Grid Middleware, (3) Grid Tools, and (4) Grid Fabric [4]. **Grid**

Fabric is consists of all the globally distributed resources that are accessible from anywhere on the Internet. These resources could be computers (such as PCs, SMPs, clusters) running a variety of operating systems (such as UNIX or Windows) as well as resource management systems such as LSF (Load Sharing Facility), Condor, PBS (Portable Batch System) or SGE (Sun Grid Engine), storage devices, databases, and special scientific instruments such as a radio telescope or particular heat sensor. **Core Grid Middleware** offers core services such as remote process management, coallocation of resources, storage access, information registration and discovery, security and aspects of Quality of Service (QOS) such as resource reservation and trading. **Grid Tools** (User-Level Grid Middleware) includes application development environments, programming tools, and resource brokers for managing resources and scheduling application tasks for execution on global resources. **Grid Applications** consists of Grid applications or portals. Grid applications are typically developed using Grid-enabled languages and utilities such as MPI (message passing interface) or Nimrod parameter specification language. An example application, such as parameter simulation or grand-challenge problem would require computational powers, access to remote data sets, and may need to interact with scientific instruments. Grid portals offer Web-enabled application services, where the users can submit and collect results for their jobs on remote resources through the Web.

II.LITERATURE REVIEW

Kumar-et al [5] proposed a non-blocking checkpointing algorithm based on keeping track of direct dependencies of processes. Each process maintains a direct dependency vector. In their scheme, initiator process collects the direct dependency vectors of all processes, computes minimum set, and sends the checkpoint request along with the minimum set to relevant processes. This reduces the time to take the checkpoints. If new dependencies are created during checkpointing process, those are updated and updated minimum set is formed.

Wang and Fuchs [6] proposed a coordinated checkpointing scheme in which they incorporated the technique of lazy checkpoint coordination into an uncoordinated checkpointing protocol for bounding rollback propagation Recovery line progression is made by performing communication induced checkpoint coordination only when predetermined consistency criterion is violated. The notation of laziness provides a trade off between extra checkpoints during normal execution and average rollback distance for recovery.

L K Awasthi-Kumar [7] proposed a minimum process coordinated checkpointing protocol for mobile distributed systems. Where the number of useless checkpoint and the blocking of processes are reduced using the probabilistic approach and by computing the tentative minimum set in the beginning. This algorithm is the first one to combine and non-blocking scheme in one algorithm.

Gupta et al. [8] proposed a single phase nonblocking coordinated checkpointing approach for mobile computing environment. In their algorithm, the processes are allowed to take the permanent checkpoints directly without taking tentative checkpoints and whenever a process is busy, the process takes a checkpoint after the completion of current procedure. However, this scheme has the disadvantage that it does consider the case of failure during the checkpointing operation which may result in the inconsistent states of the processes.

Koo-Toueg's et al.[9] proposed a minimum process blocking checkpointing algorithm for distributed systems. The algorithm consists of two phases. During the first phase, the checkpoint initiator identifies all resources with which it has communicated since the last checkpoint and sends them a request. Upon receiving the request, each resource in turn identifies all resource it has communicated with since the last checkpoint and sends them a request, and so on, until no more resource can be identified. During the second phase, all resource identified in the first phase take a checkpoint.

Silva L et al. [10] proposed Global checkpointing for all process in distributed systems. It is achieved by piggybacking monotonically increasing checkpoint number along with computational message. When a process receives a computational message with the high checkpoint number, it consider that checkpoint before processing the message. If each process allowed to initiate the checkpoint operation, the network may be flooded with control messages and unnecessary checkpoints. In order to avoid this, the proposed event allows one process to initiate checkpointing. The checkpoint event changes periodically by a local timer mechanism. When this timer expires, the initiator process broadcast checkpoint message to all other process.

J.L. Kim et al.[11] developed a new efficient synchronized checkpointing protocol which exploits the dependencies between processes in distributed systems. In this protocol, a process takes a checkpoint where all other processes agrees for same checkpoint and hence the process need not always wait for the decision made by the checkpointing coordinator as they are conventional synchronized protocols.

Prakash-Singhal et al.[12] proposed low-level checkpointing algorithm was to combine two approaches. More specifically, it forces only a minimum number of processes to take checkpoints and does not block the underlying computation during checkpointing. It forces only part of processes to take checkpoints, the carrier sense of some processes may be out-of-date, and may not be able to avoid inconsistencies. Therefore this algorithm attempts to solve this problem by having each process maintains an array to save the problem.

Chandy et al.[13] proposed a global snapshot algorithm for distributed systems. It is observed that every checkpointing algorithm proposed for message passing system. The global state is constructed by coordinating all the resources and logging the channel state at the time of checkpointing. Special messages called markers are used for coordination and for identifying the messages originating at different checkpointing intervals.

D.V. SubbaRao et al. [14] proposed checkpointing algorithm combined with selective sender based message logging. This algorithm is free from problem of lost messages. This algorithm tolerates permanent faults in the presence of other processors. In their absence it tolerates only transient failures. The term selective implies that messages are logged only within a specified interval known as active interval, thereby reducing message logging

III. FAULT TOLERANCE STRATEGIES

A fault tolerance strategy is a technique that a fault tolerant system uses either for error recovery, system repair or minimizing the impact of future failure on the overall system computation. We discuss widely-used fault tolerance strategies below.

- **Restart:** This is the simplest fault tolerance strategy. Restart resets a computation from the beginning when an error is detected. Restart can be applied locally or globally. Local restarts reset only the computation of system components that are affected by the error, while global restarts reset the entire computation of the system.
- **Check pointing:** Check pointing regularly saves the state of a system computation on a stable storage at predetermined intervals. This strategy is usually combined with other strategies like roll-back and migration.
- **Roll-back:** Roll-back is used in conjunction with check pointing. If the state of a system computation is checkpointed before an error is detected in the computation, then the computation will be rolled-back to the last stable checkpoint.
- **Roll-forward:** Roll-forward takes the computation of a failed system forward by correcting the damage that is caused by an error(s) to the overall system computation. Roll-forward is also known as forward error recovery.
- **Migration:** Migration is combined with checkpointing and roll-back. When an event that may lead to error is detected in a checkpointed system, then the computation of the system will be migrated to a new execution environment. In the new environment, the computation of the system will be restored from the last checkpoint.
- **Rejuvenation:** Rejuvenation is concerned with gradually terminating the computation of a system and then restarting or rolling-back the system immediately at potentially fault-free state. The objective of rejuvenation is to minimize the impact of transient faults on the computation of a system.
- **Replication:** Replication is concerned with simultaneously executing multiple identical replicas of a system. Replication serves two purposes: it almost guarantees at least one of the replicas will complete, and it enables voting based error detection mechanism by comparing the results of multiple replicas.
- **Redundancy:** Redundancy manages failure using primary-backup approach. Each system, usually a service or hardware, has a primary replica, and one or more backup replicas. During computation, the primary replica regularly sends its status to the backup replicas. In the event of the primary replica failure, one of the backup replicas takes the role of the primary replica. Redundancy is also known as primary-backup replication,
- **Standby spare:** Standby spare uses alternative or standby system components to ensure the computation of a system ends in success [16]. If a system component fails, then it will be replaced by a component that can carry out the functions of the failed component.
- **N-version:** In the N-version fault tolerance strategy, the function of a system is implemented using N different methods [15]. All versions of the system are executed simultaneously. Then, the outputs of all or a subset of these executions will be examined to determine whether the system completes successfully or not.

IV. RECOVERY-AWARE COMPONENT-BASED SYSTEM

A Recovery-Aware Component-Based System (RACS) is a fault tolerant grid system that realizes the RAC approach. A RACS provides fault tolerance support to RAC-based grid applications. It also provides an experiment testbed for evaluating the reliability of RAC-based grid applications. Figure 2 shows a UML [17] component diagram of a RACS reference architecture, which describes not only the structural relationship between the components of a RACS but also their mappings into a grid infrastructure. The components of a RACS are Head Manager, Compute Manager, Predictor, Injector, and Recovery-aware component. The roles of these components, their interactions, and their deployment on a grid infrastructure are discussed in subsequent sections.

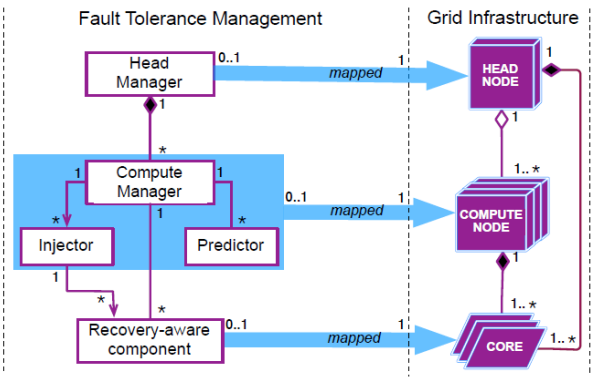


Figure 2: RACS Reference Architecture

In the RACS reference architecture, a grid infrastructure is depicted as being an execution environment with one head node and multiple compute nodes. By representing a grid infrastructure in this way, we are implying neither the compute nodes are under the management of the head node nor the compute nodes are homogeneous. The head node represents an entity, such as Xgrid controller, that is in charge of allocating the required grid resources to the activities of a grid application. A compute node represents any computing grid resource.

The roles of the components of a RACS are as follows.

Recovery-aware component: A recovery-aware component, as introduced previously, is a normal grid application

component that has an additional interface for controlling some aspects of its fault tolerance affairs.

Injector: An injector introduces simulated and real faults into recovery-aware components and their execution environment. The injector is activated if a RACS is to be used as a fault tolerance testbed. The scope of fault injection is limited by the required type of failure simulation. For simulating a host crash, for example, the injector kills all currently running recovery aware components on a given compute node. On the other hand, a CPU failure in a multi-core compute node is simulated by randomly choosing and terminating a recovery-aware component that is being executed.

Predictor: A predictor assesses the health of the currently running grid application and its environment, and then forecasts impending failures. A prediction has four possible outcomes (the sum of the probability of the prediction outcomes is 1):

- True Negative: Failure is not imminent and is predicted to be non-imminent.
- False Negative: Failure is imminent but is predicted to be non-imminent.
- True Positive: Failure is imminent and is predicted to be imminent.
- False Positive: Failure is not imminent but is predicted to be imminent.

Compute Manager: A compute manager is responsible for starting predictors and injectors, and deciding when and how to take action to recover from or proactively prevent failure. The compute manager sets the frequency of failure prediction and fault injection, and notifies injectors the scope of fault introduction (simulating node failure vs. CPU failure). When a predictor makes a positive failure prediction, the compute manager either warns affected recovery-aware components to take necessary action or executes a proactive fault tolerance strategy on their behalf. The compute manager expects regular health updates from the recovery-aware components that are under its management. If some of the recovery-aware components fail to send health updates, the compute manager marks those components as failed and executes a reactive fault tolerance strategy.

Head Manager: The head manager is responsible for starting compute managers. It also prepares detailed fault tolerance policies based on which compute managers make fault tolerance decisions. A fault tolerance policy includes the types of reactive and proactive strategies to be executed, the frequency of prediction and heart beat monitoring, and other fault tolerance related instructions. These policies can either be provided during the configuration of the head manager, system FT policy, or the submission of a grid application for execution, user FT policy.

V. FAULT TOLERANCE MANAGEMENT

The type of the fault tolerance support depends on the selected RAC architecture and fault tolerance strategy. Therefore, we identify each support using the type of the RAC architecture and the fault tolerance strategy with which the architecture is paired. For example, if the managers in MR-specific architecture handle failure using replication, then the fault tolerance support is referred to as the MR-specific replication-based RAC.

- **Restart-based RAC:** The restart-based RAC manages failure only reactively. The generic restart-based RAC restarts a failed activity whose computation does not depend on previously completed activities, whereas both the MR-specific restart-based RAC and the CL-specific restart-based RAC can restart any failed MR and CL activity, respectively.
- **Replication-based RAC:** The replication-based RAC manages failure only proactively. If an activity is predicted to fail, then the replica of the activity will be executed. If the impending failure of the activity is not predicted prior to the activity's failure, no attempt is made to recover the activity. In the replication-based RAC, at most two replicas of an activity are simultaneously executed. If a positive prediction is made while the two replicas are being executed, no more replica is instantiated even if the maximum replica limit is not reached. The generic replication-based RAC can replicate an activity only if the activity does not depend on other activities, whereas the MR-specific replication based RAC and the CL-specific replication-based RAC can replicate any MR and CL activity, respectively.
- **Check pointing-based RAC**: The check pointing-based RAC manages failure proactively and reactively. The check pointing-based RAC saves the current state of an activity whenever the activity is predicted to fail. If/when the activity fails, the activity is rolled-back to the last checkpoint. Unlike the restart and the replication counterparts, the generic check pointing-based RAC can recover the failure of any type of activity provided that specific conditions are met. If an activity that depends on a previously completed activity fails and the activity is check pointed before its failure, then the generic check pointing-based RAC recovers the failed activity. The MR-specific checkpointing based RAC and the CL-specific check pointing-based RAC can, with no restriction, manage the failure of any MR and CL activity, respectively.

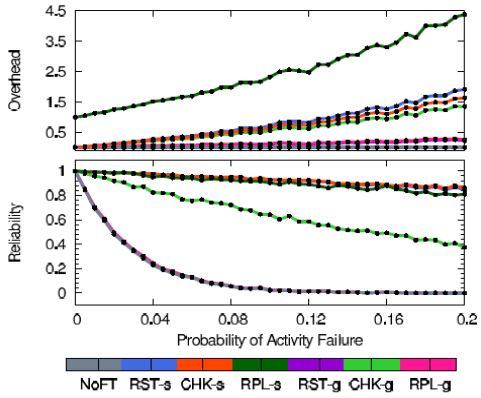
VI.RAC ARCHITECTURE EXPERIMENTS IN COMBINATIONAL LOGIC

The generic and the architecture-specific RAC approaches improve the reliability of MR and CL grid applications. Such reliability improvement, nevertheless, comes at the expense of increased cost of execution. The extent of the reliability improvement and the overhead of providing such improvement depend on the type of the RAC architecture (generic, MR-specific, CL-specific), and the fault tolerance strategy with which the RAC architecture is paired. We present the reliability improvement that MR and CL grid applications would gain by adapting the RAC.

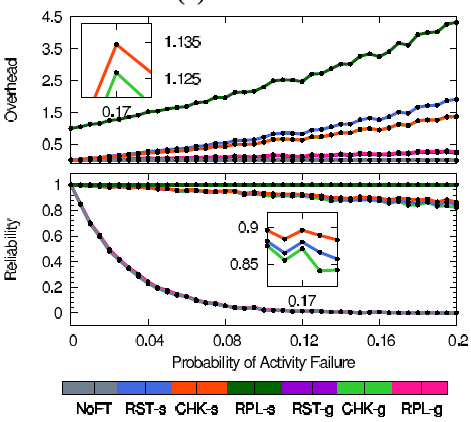
a) Restart-Based RAC

The CL-specific restart-based and the generic restart-based RAC improve the reliability of a CL application execution. The CL-specific restart-based RAC provides, as shown in

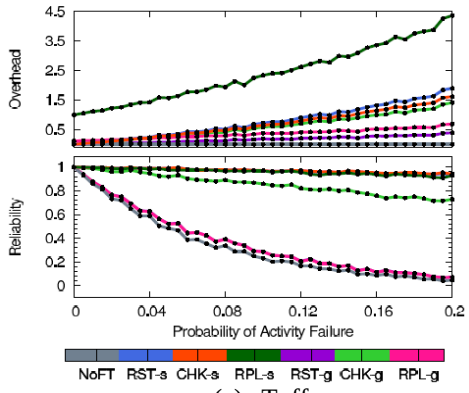
Figure 3, a more reliable execution of a CL grid application than the generic restart-based RAC. This is the result of the CL-specific RAC being able to handle the failure of any activity, and the inability of the generic RAC to manage the failure of an activity whose execution depends on previously completed computations. Hereafter we refer to an activity whose computation depends on previously completed activities as a successor activity.



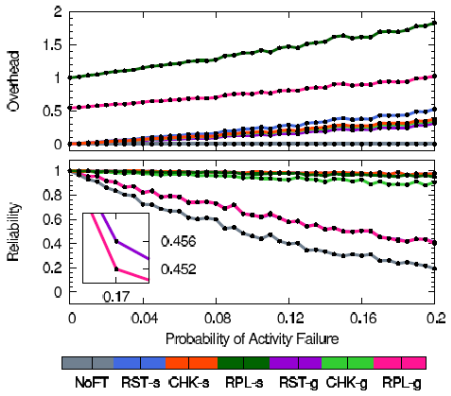
(a) FilterBank.



(b) FilterBank. Failure towards the end.



(c) Tgff



(d) Spatial Matching.

Figure 3: The reliability-overhead tradeoff of the generic and the CL-specific RAC: The inset figures magnify selected data to show the relationship between the plotted fault tolerance support types whose plots are overlapped on the scale of the outer figure.

b) Replication-Based RAC

The CL-specific replication-based and the generic replication-based RAC improve the reliability of a CL application execution. However, their performance is limited by their inability to recover a failed activity. With an increase in the probability of activity failure, as shown in Figures 3a, 3c, and 3d, more and more activities fail before their impending failure can be predicted. Unless a prediction is made, the replica of an activity will not be instantiated. Despite an increase in the probability of activity failure, as shown in Figure 3b, if a proactive strategy can be executed prior to the failure of any CL activity and the cause of the failure is a transient fault, the CL-specific replication-based RAC guarantees a 100% reliable computation.The CL-specific replication-based provides a more reliable CL computation, and introduces a higher overhead than the generic replication-based RAC. The relationship between these fault tolerance support types is similar to the one between the generic and the CL-specific variants of the restart-based RAC.

c) Check pointing-Based RAC

Both variants of the check pointing-based RAC improve the reliability of a CL application execution. The CL-specific check pointing-based RAC generally provides a more reliable CL computation than the generic check pointing-based RAC. This is due to the inability of the generic check pointing-based RAC to recover a successor activity that was not check pointed before its failure. However, if the activities of a CL application fail only towards the end of their computation or if the application is not complex, then both fault tolerance support types provide almost equally reliable CL computations. In the case when a successor activity fails only towards the end of the computation, the likelihood of the activity to have been check pointed is high. Given an activity fails after completing 95% of its computation, under the default parameter settings, where the prediction interval is 5% of a CL activity execution time and the probability of positive predictions is 0.5, there will be 19 predictions before the activity fails. Roughly half of these predictions will be positive, and therefore cause the activity to be check pointed. Once a successor activity is check pointed, the generic check pointing-based RAC can manage its failure.

The complexity of a CL application is a good indicator of the extent of the presence of successor activities in the application. As the complexity of a CL application increases, from Spatial Matching to Filter Bank, the number of successor activities in the application increases as well. The more complex the application is, the less manageable its failure will be by the generic check pointing-based RAC, and vice versa. The overhead of the CL-specific check pointing-based RAC is marginally higher than the overhead of the generic check pointing-based RAC, even when the reliability gap between the two is significant. Figure 3a, for example, shows that as the probability of activity failure increases, the difference between the two fault tolerance support types with respect to reliability increases at a faster speed than with respect to overhead. As long as an activity is check pointed, the generic and the CL-specific RAC put equivalent effort to handle its failure. Under the default parameter settings, many of the activities of the benchmark CL applications are check pointed more often than not, and thus we observe marginally equal overhead. However, due to the non-zero probability of false negative predictions, there are successor activities that will fail before they can be check pointed. As discussed previously, the presence of such activities deteriorates the overall reliability of the application whose failure is managed by the generic check pointing-based RAC.

VII. CONCLUSION

This research contributes the RAC approach, which is a fault tolerance approach that manages failure at the component level, combines reactive and proactive fault tolerance strategies, assumes runtime prediction with proactive failure management, and provides customized fault tolerance support based on the classification of the architecture of a grid application. Further, the project provides parameterized Markov models and testbed for reliability and overhead analyses. We have used the testbed for evaluating the reliability-overhead tradeoff of the RAC approach. Via simulated experiment, we have confirmed that the architecture-specific fault tolerance support provides higher reliability improvement and incurs higher overhead to grid applications than the architecture-unaware one. The degree of the reliability improvement of the architecture-specific support over the architecture-unaware one depends on factors like the type of the fault tolerance strategy selected and its parameters, and the accuracy of a predictor.

REFERENCE

[1]. Feilong Tang, Minglu Li, and Joshua Zhexue Huang. "Real-time transaction Processing for autonomic Grid applications,"Engineering Application of Artificial Intelligence 17(2004), pp.799-807, China, 2004.

[2]. Xiaolong Jin, and Jiming Liu, "Characterizing autonomic task distribution and Handling in grids," Engineering Application of Artificial Intelligence 17(2004), Pp.809-823, Hong Kong, 2004.

[3]. Rainer Unland, and Huaglory Tianfield, " Towards Autonomic computing Systems," Engineering Application of Artificial Intelligence 17(2004), pp.689-699, Germany, 2004.

[4]. Eser Kandogan, John Bailey Rob Barrett, and Paul p. Maglio, "Usable autonomic computing systems: the system administrators' perspective," In Advance Engineering Informatics 19, pp.213-221, Nov 2005.

[5]. L. Kumar, M. Misra, R.C. Joshi, "Low overhead optimal checkpointing for mobile distributed systems" Proceedings. 19th IEEE International Conference on Data Engineering, pp 686 – 88, 2003.

[6]. Wang Y. and Fuchs, W.K., "Lazy Checkpoint Coordination for Bounding Rollback Propagation," Proc. 12th Symp. Reliable Distributed Systems, pp. 78-85, Oct. 1993.

[7]. Lalit Kumar, Parveen Kumar, R K Chauhan, "Pitfalls in Minimum-process Coordinated Checkpointing protocols for Mobile Distributed", ACCST Journal of Research, Volume III, No. 1, 2005 pp. 51-56.

[8]. R K Chauhan, Parveen Kumar, Lalit Kumar, "Non-intrusive Coordinated Checkpointing Protocols for Mobile Computing Systems : A Critical Survey, ACCST Journal of Research, to be published in Volume IV, No. 3, 2006.

[9]. Pourmahmoud, S. Asbaghi, S. Haghighat, A.T. "23rd International Symposium on Computer and Information Sciences", 2008.

[10]. Koo. R. and S.Toueg..Checkpointing and Rollback-Recovery for Distributed Systems. .IEEE Transactions on Software Engineering, SE- 13(1):23-31, January 1987.

[11]. Silva L, Silva J 1992 Global checkpointing for distributed programs. Proc. IEEE 11th Symp.On Reliable Distributed Syst. pp 155-162.

[12]. J.L. Kim and T. Park. "An efficient protocol for checkpointing recovery in Distributed Systems" IEEE Transaction On Parallel and Distributed Systems, 4(8):pp.955-960, Aug 1993.

[13]. Prakash R. and SinghalM.Low-Cost Checkpointingand Failure Recovery in Mobile Computing Systems. ,IEEE Transaction On Parallel and Distributed Systems, vol. 7,no. 10, pp. 1035- 1048, October1996.

[14]. Chandy K. M. and Lamport L., "Distributed Snapshots: Determining Global State of Distributed Systems," ACM Transaction on Computing Systems, vol. 3, No. 1, pp. 63-75, February 1985.

[15]. D.V. SubbaRao and MM Naidu: A new, efficient corrdinatedcheckpointing protocol combined with selective sender based message logging, IEEE, 2008, Page(s): 444 – 447.

[16]. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. Dependable and Secure Computing, IEEE Transactions on, 1(1):11 { 33, 2004. Cited on pages 24 and 25.

[17]. P. Jalote. Fault Tolerance in Distributed Systems. Prentice Hall, Englewood Cli_, New Jersey, 1994. Cited on pages 26, 27, 28, 29, and 38.