Message Passing Fault Tolerance Design at Socket Level

Marcela Castro, Dolores Rexachs and Emilio Luque[1]

Abstract— **We present a design of a transparent fault tolerance middleware for message passing applications. The approach consists in changing the default socket model avoiding being unexpectedly closed due to a remote node failure. Moreover, a pessimistic log-based rollback recovery protocol added to this level makes possible restarting and re-executing a failed parallel process until the point of failure independently of the rest of the processes. All this work is made automatically and in a transparent way for the application. This service can be optionally activated at runtime at user level. The models used for protection and recovering application and detection of failures are based on RADIC architecture. We have tested this middleware by executing a master-worker (M/W) and Single Program Multiple Data (SPMD) applications which follow different communication patterns.**

Keywords— **Fault-tolerance; High-Availability; RADIC; message passing; socket**

I. Introduction

NOWADAYS the need of having fault tolerance (FT) solutions for parallel applications is indisputable and they are regarded as a mandatory requirement for executing critical applications which last more than the MTBF of the host cluster. The risk of suffering a stop due to a node failure is too high for not taking preventive measures.

Although Fault Tolerance in High Performance Computing (HPC) systems has been extensively researched in recent years offering different approaches, there is not yet a FT solution where the failures are completely hidden from the application avoiding any modification at user's code.

When a message passing application is executing in a cluster and suddenly one of the node fails, the communications established with the parallel processes in it also fall down. These communication errors would propagate causing fatal errors to the rest of the parallel processes.

The Fig. 1 shows a typical communication level diagram of a message passing application. A failure at physical o networking levels usually spreads up errors to higher levels causing an undesirable execution stop of the application.

Socket is a *de facto* standard API of POSIX Operative Systems to use the transport level protocols like TCP or UDP. This API is normally used for interchanging data packages between two executing processes in a cluster. The socket model is intended to do that then, it does not provide a failure model able to recover the connection from fatal node failures. However, controlling socket errors caused by a fall of remote peer would prevent the propagation of them to the upper levels of the message passing communication library and application.

The research work *Reliable Network Connections* [1] describes *rocks*, an approach which changes the normal behaviour of the diagram state of socket API. The new API automatically detects network connection failures, including those caused by link failures, extended periods of disconnection, and process migration, within seconds of their occurrence. When this kind of communication error happens, instead of closing unexpectedly the socket, the IP address is replaced by the new location of remote peer and the broken connection is recovered without loss of in-flight data as connectivity is restored.

Clearly, establishing reliable network connections instead of normal ones would contribute to provide a FT solution for message passing application avoiding unexpected fatal errors.

Message passing applications usually rely on rollback-recovery protocols to recover from failures. Most of these protocols were explained and classified by E.N. Elnozahazy [2]. RADIC *Redundant Array of Distributed Independent Controllers* [3] is a fault tolerance architecture for message passing applications that defines a proper model to apply a rollback recovery protocol using uncoordinated checkpoint and pessimistic log-based on receiver.

The FT strategy of this research work basically consist in modifying the socket model used by message passing communication library and parallel application (upper levels indicated in Fig. 1). The new model combines the use of reliable network connections with the models of RADIC architecture.

The aim is assuring the execution ends successfully in spite of node failure for whatever message passing library is in use and avoiding making changes in the program or in the library. Moreover, we consider important to let the user choose whether the execution uses FT or not.

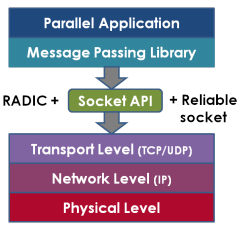


Fig. 1. Socket Level

II. Related Works

The approaches used to add FT in a message passing application can be classified in three groups ac-

[1]Computer Architecture and Operating Systems Department,Universitat Autònoma de Barcelona, mcastro@caos.uab.es; {dolores.rexachs; emilio.luque}@uab.es

cording to [4]. First, the application can be changed adding the FT mechanisms. In relation to this approach, we can mention research works like [4] or [5] which give solutions that facilitate the programming tasks either defining FT programming patterns or adding new libraries to be called. Although this first group of FT solutions is likely to reach the best fit, the programing cost is high and not always applicable if the source code is not available. In the second group we can categorize the research works which locate FT algorithms in communication library. Most of the used solutions belong to this group, because the application does not need to be changed. We regard this kind of tools as an extension of the MPI communication library. MPICH-V Project [6] is an example of this case. Moreover, RADIC was previously implemented using this kind of strategy [3]. Although the application is not changed, it needs to be compiled again with the modified communication library. that could be a problem if we only have the executable programs. Another drawback of this group is the need of adapting each MPI implementation to the specific FT strategy. Finally, available solutions at system level are also transparent for the application, but, most of them are very sensitive to changes in operative system versions and they are most difficult to adapt to other architectures. An example using this last category is DMTCP [7], a checkpoint and restart tool for distributed applications which can be also used for message passing applications. However, scripts for checkpoint and restarting must be provided by the user. Our work fits in this last category as it works at system level.

On the other hand, there are three requirements to be covered by rollback recovery FT approaches. First, **Protection** of information/state to continue computation. Second, **Detection** of the failure and last, **Restart** the computation reconfiguring the system to isolate the damage component and mask the errors. Most FT solutions are not fully developed with all the requirements at system level.

For example, BLCR [8] is a well-known project of kernel-level process checkpoint. It can be used with multithreading programs but it does not support distributed or parallel process. This tool covers the protection and restarting requirements. The detection has to be added by the user.

DMTCP [7] (Distributed Multithreaded Checkpointing) does not provide the detection requirement to be considered a FT solution.

DejaVu [9] is a transparent user-level fault tolerance system for migration and recovery of parallel and distributed applications. It provides the three mentioned requirements and implements a novel mechanism to capture the global state named online logging protocol. Although uncoordinated checkpoints are performed, it uses a coordinated mechanism to assure the global consistent state of them. This property can be a drawback to scale properly. In addition, DejaVu does not implement any log message protocol, so all parallel processes are forced to recover and restart in case of a node failure.

RADIC [3] model meets the requirements of detection, protection and recovery. The behavior is completely distributed on the nodes of the clusters and the overhead added during the protection phase and in recovery is independent from the number of processes. This property is essential nowadays when the numbers of processors in the clusters are increasing so much. To protect it uses log message based on receiver rollback recovery protocol which facilitates the recovery tasks, but adding some overhead during the protection phase. The middleware we are presenting is based on RADIC and works at user level.

III. Design Requirements

This section defines the two basic requirements to take into account during the design of the middleware. The first is that the design has to be located at socket level in order to achieve application and library independence. The second requirement is having properties of transparency, distributed, decentralization and scalability, which are going to be inherited from RADIC. This section begins with a brief explanation of RADIC architecture. Previous research papers can be consulted for detailed information [3]. Finally, the concept of reliable sockets is defined, outlining how we can include them and what is required to do it.

A. Radic Architecture

RADIC architecture[1] is based on uncoordinated checkpoints combined with pessimistic log-based on receiver. Critical data like checkpoints and message logs of one application process are stored on other node different from the one in which the parallel process is running. RADIC defines the following two components also depicted in Fig. 2

- **Observer (Oi):** this entity is responsible for monitoring the application's communications and masks possible errors generated by communication failures. Therefore, the observer performs message logs in a pessimistic way as well as it saves periodically the parallel process state by checkpointing. Message logs and checkpoints are sent to protector **Ti-1**. There is an observer **Oi** attached to each paralell process **Pi**.
- **Protector: (Ti)** There is one running on each node which can protect more than one application process. In order to protect the application's critical data, protectors store that on a non-volatile media. In case of failure, the protector recovers the failed application process with its attached observer. Protector uses heartbeat/watchdog mechanism to detect neighbour failures and recover and reconfigure in these cases.

B. Reliable sockets

TCP is a reliable transport protocol between two peers in a sense that every packet sent by one peer is assured to be delivered and received by the other peer respecting the sent order. Each peer uses send

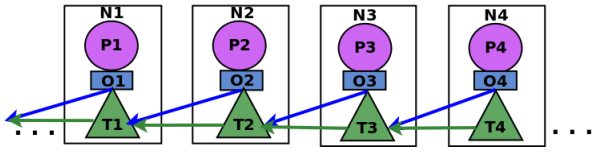


Fig. 2. RADIC diagram - Diagonal arrows: **Oi** sends the critical data to its protector **Ti-1** - Horizontal arrows: Protector **Ti** sends heartbeat signal to **Ti-1**

and receive buffers managed by flow-control to accomplish this reliability.

However, the TCP failure model does not provide a mechanism to recover the connection from a permanent failure of one of the peers, because this kind of situation is out of the scope of a transport protocol. Usually, the applications running on POSIX Operative System use socket API as I/O network interface to receive and send data through a TCP/IP connection. TCP connection failures occur when the kernel aborts a connection. This could be caused by several situations such data in the send buffer goes unacknowledged for a period of time that exceeds the limits on retransmission defined by TCP, or receiving a TCP reset packet as a consequence of the other peer reboots or closes the socket unexpectedly. Furthermore, when the kernel aborts the connection, the socket becomes invalid for the application.

If the application does not have a proper functionality to recover this invalid socket, usually the execution is aborted due to the unexpected situation.

Rocks architecture proposed by V.Zandy [1] defines the operation of a reliable socket by changing the default state socket diagram by a new one. This new operation does not allow the socket to be closed by such exceptional situation. Instead of that, the socket remains in a suspended state while a new address of the remote peer is got and the socket is reconfigured. This new socket behavior affects the process, not the internal TCP socket state maintained by the kernel.

This general idea is applicable to our design, but not the detailed behavior and implementation. *Rocks* is applied to distributed and mobile applications and not for message passing ones.

We need to incorporate to this socket level the functionality of the rollback recovery protocol defined by RADIC models. To accomplish this task, we designed a new behavior model at socket level considering reliable sockets and pessimistic based on receiver rollback recovery protocol.

IV. Design

Following the basic idea of reliable network connections, the FT logic needed for protection and for restarting a process is added by interposing socket functions as *socket, bind, listen, connect, send* and *recv*. The default socket state is changed for not allowing the socket be closed when remote peer falls down. Next, the socket does not become invalid for the upper level process. RADIC recovery model determines that the failed processes are restarted in the protector node. As a result, the observer is able to reconfigure the socket with this new address and then, the lost connection with the restarted process is re-established.

RADIC components are involved in the three well differenced functional phases: protection, detection and recovery. The next subsections explain how these components behave to accomplish these phases.

A. Protection Phase

The tasks related to save the consistent state of the application are considered into this functional phase of protection. The observers are responsible for doing checkpoint, interpose receiving messages and send them to their protectors. The Fig.3 shows the connection **(1)** between each observer **(Oi)** with its protector **(Ti-1)** used to send this critical data. Observers are also in charge of building the reliable connections. A parallel canal **(2)** with the same characteristics is opened for each socket established by the application. This canal, named **control-ft socket**, is used by the two peers observers connected in order to interchange their process identification and acknowledgement data. **Control-ft socket** is used during Message log, Checkpointing and Restarting which are detailed later in next Section V in subsections V-A, V-B and V-C respectively.

Protectors collaborate in this functional phase receiving critical data from their observers and save it in stable storage **(1)**.

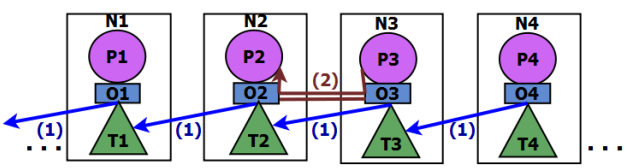


Fig. 3. RADIC Protection Model

B. Detection and Recovery Phases

The detection is carried out by protectors sending a heartbeat to their predecessor neighbour. In Fig. 4 the line numbered with **(3)** represents the connection of each protector with its protected node. On the other hand, when the observer detects a communication error while a socket is sending or receiving packages, the observer executes the next diagnostic procedure. First of all it tries a configurable number of times to reconnect the socket. If the connection was lost due to a checkpoint of the remote peer, it is likely that it is reconnected in this first retrying, as it was described previously in protection phase IV-A.

In case the observer could not reconnect, it asks to the protector of the failed peer for the state of the process. The protector answer could be that the node where the process is executing is still alive, or that the process is doing checkpoint or that it is in recovering due to its node host has a failure. If one of the first two options are received, the observer keeps recovering the lost connection retrying as it was done in the first stage of this procedure. The third answer includes the new IP address of the recovering process,

so the socket is reconfigured before retrying. When the remote process is recovered, the connection is established after this stage. Lastly, the failed *send* or *recv* operation is launched again.

As an example of the diagnostic procedure, Fig. 4 shows the detection and recovery done if **N3** fails. The observer **O4** connected to such node, after retrying unsuccessfully on reestablishing connection, asks for the state to the protector using **(4)**. Protector of **N3** (**T2**) detects node failure using heartbeat or by receiving an diagnostic from an observer. Meanwhile, protector **T4** establishes **(5)** with **T2** to confirm that connection with **T3** is lost. As a result, **T2** recovers and restarts **P3** and **O3** in **N2**. **O3** reads message log previously saved by **T2** using**(6)**, looks for a new protector **T1** and sends it a new checkpoint. Finally, **N4** is assigned to a new protector **T2**. Consequently, **T4** sends hearbeat signal to it by using **(3)** and the **O4** sends its critical data to **T2** by using **(5)**.

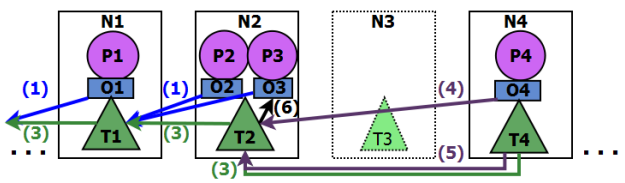


Fig. 4. RADIC Detection and Recovery Model

V. RADIC using Secure Sockets

Taking into account that RADIC defines that the **observer** component is that one attached at each parallel process, the interposition library at socket level corresponds to this. Consequently, the library has to accomplish all the functionality of this component defined by RADIC models. This paper is specially focused on defining this entity, because it is directly affected by the approach adopted. In contrast, the **protector** can be seen as an independent process that only interacts with other RADIC components like observers and other protectors. The functionality of protectors is completely defined, tested and explained in previous research works.

This section describes the three pieces of functionality needed to be incorporated at reliable socket level to get a RADIC **observer.** These pieces are message log, checkpointing and restarting. Each of them presents different challenges to face, which are explained in the following subsections including the way they are overcomed.

A. Message Log

A pessimistic log-based on receiver rollback-recovery protocol has to be designed at socket level in order to assure that the state of each process is always recoverable. This kind of procedure can add some overhead during the normal execution (protection phase) but this way simplifies the recovery tasks because the effects of a failure are confined only to the processes that need to be restarting.

Log-based rollback-recovery assumes that all nondeterministic events can be identified and their corresponding determinants can be logged to stable storage. Receiving a packet is considered a nondeterministic event to log.

At first sight, it seems a simple challenge that can be solved interposing *recv* function and sending the received message to the protector afterwards.

But pessimistic logging protocols are designed under the assumption that a failure can occur after any nondeterministic event in the computation. This assumption is "pessimistic" since in reality, failures are rare. This property stipulates that if an event has not been logged on stable storage, then no process can depend on it. For this reason, a sender of a message waits an ack which indicates the message is completely saved in stable storage. To accomplish this requirement of acknowledgment of each received and saved package, a communication between the two **observers** involved in each peer is established, named **control-ft socket**. We cannot use the application socket being interposed to send and receive acknowledge data because we can be interfering on the application protocol affecting the integrity of their messages.

Therefore, for each socket established by the upper level, the interposing library creates a new socket named **control-ft socket** used to interchange control data between two observers intercepting *send* and *recv* functions.

The Fig. 5 shows how a message is treated since it is generated from the sender process. The *send* operation is interposed by sender observer **Os** which sends a numerated acknowledgement requirement to the receiver observer using the **control-ft socket** canal, represented by dotted lines. The message is sent to the receiver using the **real socket**, depicted as solid lines. The receiver observer **Or** interposes the *recv* operation and receives the acknowledgement requirement through the **control-ft socket** and the application message through the **real socket**. Observer **Or** sends the message to its protector. Once **Or** receives the ack of save operation, sends the ack to sender and delivers the message to process finishing the *recv* interposed. Observer **Os** receives the ack indicating this message is correctly saved and it is not necessary to be resent anymore. Lastly, the *send* interposition is finished and the process resumes the processing. The gray block represents the tasks added by the logging message protocol during failure-free operation.

B. Checkpointing

Each parallel process has to be checkpointed periodically in order to save its state. In a log-based protocol, checkpointing is performed in order to limit the amount of work that has to be repeated in execution replay during recovery. This task is performed in an uncoordinated way, thus no centralized or blocking mechanisms are needed in the sake of scalability.

During the checkpoint, all the active communi-

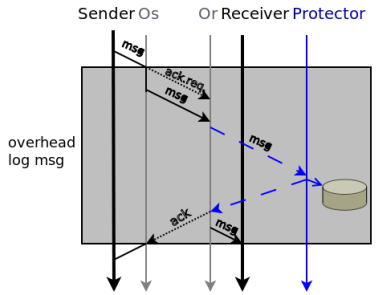


Fig. 5. Message log: Real sockets: Solid lines - Control-Ft socket: dotted lines - RADIC sockets: dashed lines

cations of the observed parallel process need to be closed. The BLCR library being used to checkpoint processes recommends this procedure for two reasons [8]. First, to avoid loosing in transit data, and second, because the socket and its corresponding connections have to be established again from scratch during the restarting in a new cluster node.

Therefore, all the opened sockets have to be closed before checkpointing and re-opened and re-established after it. To accomplish this task we need to keep the following data as it is not provided by the operative system:

- **Virtual socket**: It is the socket number handler known by the parallel process get it during a *socket* or *accept* function.
- **Socket Type:** This type can be *connect, accept* or *listen*. It is used to identify which operation has to be performed to re-establish the socket after checkpoint or during restart.
- **Re-establish parameters:** The parameters used originally to execute the function *connect, accept or listen* interposed in order to re-execute the command after checkpointing or restarting.
- **Real socket**: Socket number actually being in use to intercommunicate with remote process, getting during a re-open operation after checkpointing or restarting. The operative system delivers different id socket handler when a *socket* or *accept* function is re-executed. A one-by-one relation between virtual and real socket number is kept. During the interposition the observer changes the virtual socket id referenced by the application by this real socket number. Therefore, the function is performed using the current real opened socket.

There is a problem to be solved when an *accept* type socket is re-established. During *accept* re-execution multiple observers clients can be trying to re-connect at the same time. The server observer has to be able to recognize which is the other peer in order to continue the control message logging of the broken socket. Using operative system available functions, the remote ip and port can be known but this data is not enough to identify uniquely the observer client process previously connected to this socket due to more than a parallel process can be executing in one node.

To accomplish this identity validation task, each observer sends a unique parallel process identification **(pid)** trough the **control-ft socket**. Consequently, during the re-establishment of an *accept* socket the former client connected can be uniquely identified to continue the log message associated to the virtual socket opened by the application.

For instance, a master accepts connections coming from two workers. They are connected using sockets 4 and 6 respectively. The socket 4 has socket 5 as **control-ft** and the 6 has the 7. The sockets are closed before checkpoint. After finishing checkpointing, the *accept* functions are executed to re-establish connections. But two observers associated to the two workers are trying to re-connect the unexpectedly closed socket at the same time. After connecting, the remote observer sends its identification using the **control-ft**. In this way, the server can determine which real socket corresponds with virtual socket 4 and which with the 6.

C. Restarting

When a node fails, the protector recovers the processes which were being saved in it using the last checkpoint received. Each process is restarted with its corresponding observer. This observer detects the restarting state and its behavior is different until the process arrives to the point of failure. This point is reached when the last received message in log is consumed by the restarting parallel processes.

Two important design decisions were made in order to get RADIC restart model at socket level. First of all, the sockets type *listen* which were active on failed host, are launched on this new host. These sockets are needed to be ready before re-connecting the sockets type *accept* being re-executed. In the second place, to re-execute the parallel process until the point of failure, the observer in restarting mode, intercepts the *recv* function and the contents is extracted from the log message previously saved by the protector.

VI. Experimental Results

We test the fault tolerance middleware in order to validate the functionality of protection and recovery phase. The principal aim is to assure that the mechanism used to build the reliable tunnel connections and the log message protocol are working correctly. Our second aim is to know the overhead added in execution time by protection and recovery processes.

The experiments were executed on a cluster formed by 4 nodes Intel® Core™ i5-650 Processor 6GB RAM, Network Gigabit Ethernet. The OS used is Ubuntu 10.04 Kernel 2.6.32-33-server.

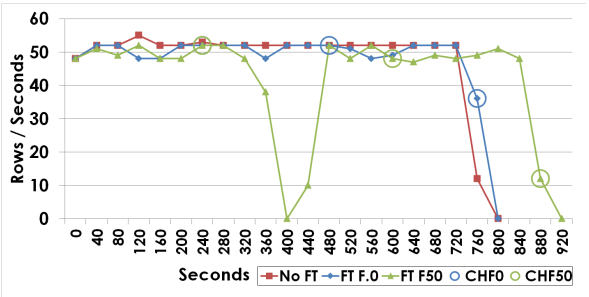
We use a sum of matrices Master/Worker and a heat-transfer SPMD applications based on TCP sockets, which follow different communication patterns, in order to test the interconnection socket model performed after checkpoints and in restart.

We use three ways of execution. First, the normal without FT *No FT*, second using FT but without

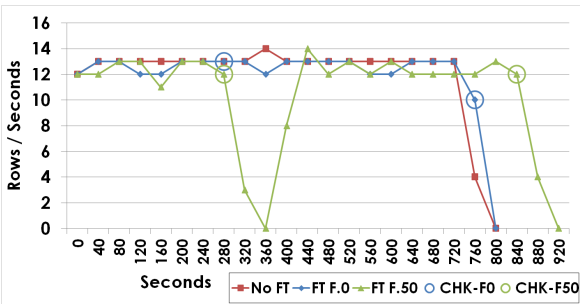
any node failure *FT F0* and lastly, we inject a fail in the process executing on the **N3 (P3)** some events after the first checkpoint, 50 in M/W *FT F50* and 100 in SPMD *FT F100*.

Two selected experiments are shown in Figs. 6 and in Fig. 7 depicting the throughput (rows/seconds or iterations/seconds) achieved during the three-way executions.

The M/W was executed with 4 workers, one per node. Fig. 6(a) shows master execution and Fig.6(b) represents one of the worker process where the failure is injected. In all cases, the red line *No FT* is slightly upper the blue one *FT F0*. In MW experiment 2.61% of overhead in execution time was added by the protection system. The fall was caused by the failure. The master throughput is affected some time after the worker falls, while the worker is restarting, processing the log message. The worker shows a maximum throughput due to the re-executing procedure using local log message. The total execution time of *FT F50* was of 888seg, 15.78% more than the original. The circles show the two checkpoints triggered during *FT F0* and *FT F50* executions.



(a) MW Master process



(b) MW Failed Worker process

Fig. 6. Master/Worker evaluation results.

The heat-transfer SPMD was executed with 4 processes, one per node. Fig. 7 graphs the execution of the process chosen to inject the failure during *FT F100*. The overhead of executing with FT but without failure was of 3.24%. When a fail is injected *FT F100* the total execution time was of 6.76% compared with the normal execution without FT *No FT*. The circles show the 4 checkpoints performed during *FT F0* and *FT F100* executions.

VII. Conclusions and Future Work

The results show that it is possible to build a transparent FT system at reliable socket level able to provide no fail stop to message passing application independently from the communication library. In that way, the user is not forced to choose a specific communication library to get the fault tolerance facilities and the library may be chosen.

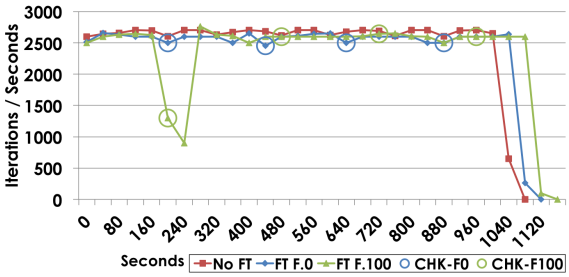


Fig. 7. SPMD evaluation results

We are working on a set of experiments to prove we can use this middleware to fault tolerance applications using either MPICH or Open-MPI.

Future work will include as well an analysis of the scalability of the solution, testing if the speedup of applications being protected is kept in spite of using fault tolerance.

Acknowledgments

This research has been supported by the MICINN Spain under contract TIN2007-64974, the MINECO (MICINN) Spain under contract TIN2011-24384, the European ITEA2 project H4H, No 09011 and the Avanza Competitividad I+D+I program under contract TSI-020400-2010-120.

References

[1] Victor C. Zandy and Barton P. Miller, "Reliable network connections," in *Proceedings of the 8th annual international conference on Mobile computing and networking*, New York, NY, USA, 2002, MobiCom '02, pp. 95–106, ACM.

[2] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput.Surv.*, vol. 34, no. 3, pp. 375–408, September 2002.

[3] Leonardo Fialho, Guna Santos, Angelo Duarte, Dolores Rexachs, and Emilio Luque, "Challenges and issues of the integration of radic into open mpi," in *16th European PVM/MPI Users' Group Meeting on Recent Advances in PVM and MPI*, 2009, pp. 73–83.

[4] William Gropp and Ewing Lusk, "Fault tolerance in message passing interface programs," *Int. J. High Perform. Comput. Appl.*, vol. 18, pp. 363–372, August 2004.

[5] Sriram Rao, Lorenzo Alvisi, Harrick M. Viny, and Department Computer Sciences, "Egida: An extensible toolkit for low-overhead fault-tolerance," in *In Symp. on Fault-Tolerant Comp.* 1999, pp. 48–55, Press.

[6] Aurelien Bouteiller, Thomas Hrault, Graud Krawezik, Pierre Lemarinier, and Franck Cappello, "MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI," *IJHPCA*, vol. 20, pp. 319–333, 2006.

[7] Jason Ansel, Kapil Arya, and Gene Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *IPDPS*, 2009, pp. 1–12.

[8] Paul H. Hargrove and Jason C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, pp. 494, 2006.

[9] Joseph F. Ruscio, Michael A. Heffner, and Srinidhi Varadarajan, "Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 119, 2007.