RADIC-based Message Passing Fault Tolerance System

Marcela Castro, Dolores Rexachs and Emilio Luque
Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona
marcela.castro@caos.uab.es; {dolores.rexachs; emilio.luque}@uab.es

***Abstract*—We present an analysis design of how to incorporate a transparent fault tolerance system at socket level for message passing applications. The novel design changes 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. This paper explains and analyzes the design time decisions. We tested and assessed them executing a master-worker (M/W) and Single Program Multiple Data (SPMD) applications which follow different communication patterns. Promising results of robustness in interprocess communication were obtained.**

Keywords-Fault-tolerance; High-Availability; RADIC; message passing; socket.

I. INTRODUCTION

Fault tolerance (FT) solutions are regarded as a mandatory requirement for parallel applications since the probability of failure is higher in increasingly complex High Performance Computing (HPC) systems and with more components. There is a high risk of suffering an execution stop due to a node failure when the parallel application lasts more than the Mean Time Between Failures (MTBF) of the host system.

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 Figure 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 [1] is a *de facto* standard application interface (API) of Portable Operating System Interface (POSIX) to use the transport level protocols like TCP or UDP [2]. This API is normally used for interchanging data packages between two executing processes in a cluster. The socket model is intended to do that, but a remote failure is treated by this API as a fatal error. 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* [3] describes *rocks*, an approach which changes the normal behaviour of the diagram state of socket API by automatically detecting 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 [4]. RADIC *Redundant Array of Distributed Independent Controllers* [5] 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 approach of FT of this research work basically consist in modifying the socket model used by the upper levels indicated in Figure 1. The new model combines the use of reliable network connections with the models of RADIC architecture in order to provide a FT system for parallel application that would be used independently of what message passing library is in use. This independence let the FT be seen as an additional optional infrastructure service without requirements for the application.

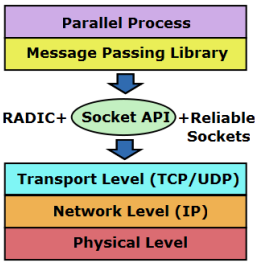


Figure 1. Socket Level

This paper is specially focused on explaining the requirements of the system, the problems that have fulfilled each of them and the corresponding solutions adopted in the design of a RADIC-based fault tolerance system at socket level.

A RADIC-based FT system inherits its properties of dis-

 ISBN: 978-1-61208-237-0

tributed and decentralized, which would facilitate to develop a scalable solution.

The content of this paper is organized as follows. In Section II we mention the related works. Section III defines the requirements to take into account in the design of this solution. The Section IV explains the problems and the solutions adopted to fulfill the requirements at socket level. The experimental evaluation is presented in Section V, and lastly, we state the conclusions and the future work in Section VI.

II. Related Works

The approaches used to add FT in a message passing application can be classified into three groups according to [6]. First, the application can be changed adding the FT mechanisms. In relation to this approach, we can mention research works like [6] [7], which 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, it is expensive 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 [8] is an example of this case. Moreover, RADIC was previously implemented using this kind of strategy. [5]. Although the application is not changed it needs to be compiled again with the modified communication library. Furthermore, this could be a problem if we only have the executable programs and we still need to assure an error-free execution in spite of node-failures. 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 not easily portable to other architectures. An example using this last category is DMTCP [9], 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. Firstly, **Protection** of information/state to continue computation. Secondly, **Detection** of the failure and lastly, **Restart** the computation reconfiguring the system to isolate the damage component and mask the errors. Fault tolerance solutions are not fully developed with all the requirements at system level.

For example, BLCR [10] 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 [9] (Distributed Multi-threaded Checkpointing) does not provide the detection requirement to be considered a FT solution.

DejaVu [11] is a transparent user-level FT 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 [5] meets the requirements of detection, protection and recovery. These tasks are carried out without any centralized element to keep the scalability of the running application. Thus, 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. We think 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 FT system. 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, distribution, 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 [5] [12]. 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 is based on uncoordinated checkpoints combined with pessimistic log-based on receiver. Critical data like checkpoints and message logs of each parallel process are stored on a different node from the one in which it is running. This selection assures application completion if a minimum of three nodes are left operational after *n* non-simultaneous faults. In short, RADIC defines the following two components also depicted in Figure 2

 ISBN: 978-1-61208-237-0

- **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 detects node failure by sending heartbeats to its neighbours.

Figure 2 shows the relationship between nodes running an application with RADIC fault tolerance architecture. Diagonal arrows represent critical data flow while horizontal ones represent heartbeats.

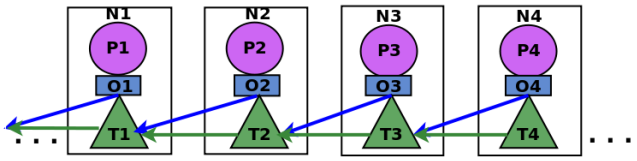


Figure 2. RADIC diagram shows each observer **Oi** sends the critical data to its protector **Ti-1**. Each protector **Ti** sends heartbeat signal to **Ti-1**

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

However, the TCP 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 [3] 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, because they considered only two peer applications and not parallel ones composed by several processes.

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

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 diagram state of socket API is changed in order to not allow the socket to 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 re-configure the socket with this new address and then, the lost connection with the restarted process is re-established.

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. As we mentioned before, 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.

IV. Fault tolerance using Secure Sockets

This section describes the three pieces of functionality needed to be incorporated at reliable socket level to get a RADIC **observer.** These three 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 overcame.

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

 ISBN: 978-1-61208-237-0

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. Because of that, a sender of a message needs to wait until the complete sent message is saved in stable storage before continuing its operation. Once a received message is completely saved on stable storage, an acknowledgment is sent to the sender.

To accomplish this requirement of acknowledgment of each received and saved package, we need to establish a communication between the two **observers** involved in each peer of a 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 Figure 3 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 finishes 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 the failure-free execution.

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.

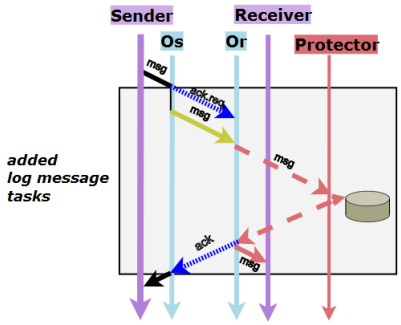


Figure 3. Message log: Real sockets: Solid lines - Control-Ft socket: dotted lines - RADIC sockets: dashed lines

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 communications of the observed parallel process need to be closed. The BLCR library being used to checkpoint processes recommends this procedure for two reasons. 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 id known by the parallel process achieved 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 id 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 an *accept* re-execution, multiple client observers can be trying to re-connect at the same time. The server observer has to be able to recognize

 ISBN: 978-1-61208-237-0

which is the other peer in order to continue the control message logging of the broken socket. Using operative system 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. The processes are recovered in a spare node if there is one available or in the same protector node if there is not. 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.

V. Experimental Results

We test the fault tolerance system for validating the functionality of RADIC and reliable connection at socket level. The principal aim is to assure that the mechanisms 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. We take some measures to have indicators for assessing how the system is working in terms of overhead and bandwidth consumed by the protection model.

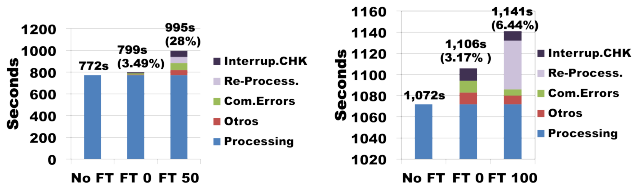
The experiments were executed on a cluster formed by 4 nodes Intel® Core™ i5-650 Processor 6GB RAM, Network Gigabit Ethernet. The OS used is Ubuntu 10.04 Kernel 2.6.32-33-server.

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 do a better test of the reliable socket model performed after checkpoints and in restart process.

We use three ways of execution. First, the normal without FT **No FT**, second using FT but without any node failure **FT 0** and lastly, we inject a fail in the process executing on the node **N3** some events after the first checkpoint, 50 in M/W **FT 50** and 100 in SPMD **FT 100**.

The M/W was executed with 4 workers, one per node. The first node executes the master and one worker. Master performs 2 checkpoint of 450Kb(avg) and workers 3 of 425kb(avg). SPMD is executed with four processes, also one per node. Each one executes 3 checkpoints of 1440kb(avg). The fall is injected in **N3** on both cases and the worker or spmd process are recovered in **N2.**

Two selected experiments are shown in Figure 4(a) and in Figure 4(b). The diagrams show the overhead time comparing the three ways of execution. These times are measured in the process executing in **N3.** The total execution is divided into: the seconds used by interruptions to perform checkpoints, time used for restarting and re-executing, seconds for recovering from communication errors due to node failures or remote checkpointing, time used by other reasons that are not measure by now, like detection of errors, and finally the base time that the application last without FT **(No FT).** The



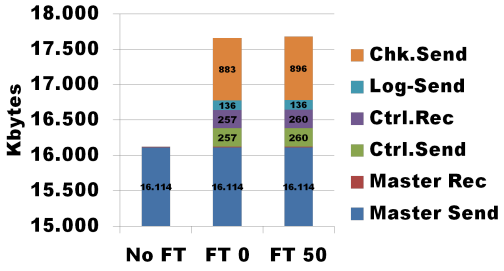
(a) MW Time overhead (b) SPMD Time overhead

Figure 4. Experimental results time execution results.

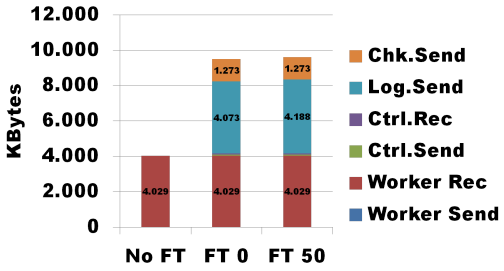
Figure 5(a) graphs the traffic sent and received by process master during the tree executions. and Figure 5(b) shows the same for the worker executed in **N3** that was directly affected by the failure. Master sent two checkpoint to protector. As the master receives very few data, the bytes sent to protector to log message is few. The data transfered by **control-ft** canal is proportionally low. On the other hand, the worker receives much more data, therefore, more kbytes of log are sent to protector.

Finally, in Figure 6, the traffic of the SPMD process executing in **N3** is shown. Similarly as it was observed

 ISBN: 978-1-61208-237-0



(a) MW Master process



(b) MW Failed Worker process

Figure 5. Master/Worker traffic overhead analysis

in previous executions, the overhead added by **control-ft** is low, and the log message is directly proportionated with the amount of received data.

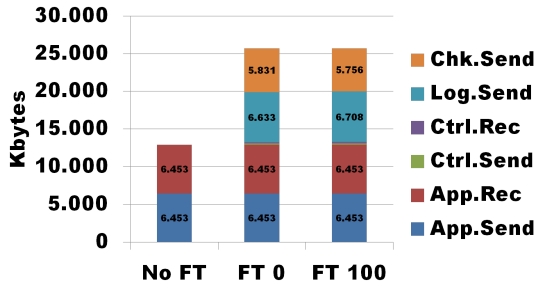


Figure 6. SPMD traffic overhead analysis

VI. Conclusions and Future Work

The results show that the design made of a transparent and distributed fault tolerance system is appropriate. Message passing applications with different communication patterns are able to end successfully performing periodic checkpointing and restart and re-execute in case of node failure.

Using this approach it is possible to build a transparent fault tolerance middleware able to provide no fail stop to message passing application independently from the communication library in use. In that way, the user is not forced to choose a specific communication library to get the fault tolerance facilities. The library of preference can be chosen.

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

Future work will include as well an analysis of the scalability of the middleware, testing if the speedup of applications being protected can be 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 contract TIN2011-24384, the European ITEA2 project H4H, No 09011 and the Avanza Competitividad I+D+I contract TSI-020400-2010-120.

References

[1] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The design and implementation of the 4.4BSD operating system*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1996.

[2] R. Gordon, "The tcp/ip guide: A comprehensive, illustrated internet protocols reference." *Library Journal*, vol. 131, no. 1, pp. 146–146, JAN 2006.

[3] V. C. Zandy and B. P. Miller, "Reliable network connections," in *Proceedings of the 8th annual international conference on Mobile computing and networking*. New York, NY, USA: ACM, 2002, pp. 95–106.

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

[5] L. Fialho, G. Santos, A. Duarte, D. Rexachs, and E. 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.

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

[7] S. Rao, L. Alvisi, H. M. Viny, and D. C. Sciences, "Egida: An extensible toolkit for low-overhead fault-tolerance," in *In Symp. on Fault-Tolerant Comp.* Press, 1999, pp. 48–55.

[8] A. Bouteiller, T. Hrault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI," *IJHPCA*, vol. 20, pp. 319–333, 2006.

[9] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *IPDPS*, 2009, pp. 1–12.

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

[11] J. F. Ruscio, M. A. Heffner, and S. Varadarajan, "Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems," *Parallel and Distributed Processing Symposium, International*, p. 119, 2007.

[12] G. Santos, A. Duarte, D. Rexachs, and E. Luque, "Providing non-stop service for message-passing based parallel applications with radic," ser. Lecture Notes in Computer Science, vol. 5168 LNCS, 2008, pp. 58–67.

 ISBN: 978-1-61208-237-0