J. Parallel Distrib. Comput. 86 (2015) 98–111

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Fault tolerance at system level based on RADIC architecture

Marcela Castro-León *, Hugo Meyer, Dolores Rexachs, Emilio Luque

Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona, Campus UAB, Edifici Q, 08193 Bellaterra (Barcelona), Spain

HIGHLIGHTS

- A system-level fault-tolerant mechanism for message passing applications.
- Fully decentralized and transparent, for applications and communication library.
- Protection, detection and recovery functions, implemented at socket API level.
- Semi-coordinated vs. uncoordinated checkpoints: performance based election.

ARTICLE INFO

Article history:
Received 18 February 2015
Received in revised form
23 June 2015
Accepted 20 August 2015
Available online 28 August 2015

Keywords:
Software fault tolerance
Resilience
RADIC
Message passing
Semi-coordinated checkpoint
Uncoordinated checkpoint
Socket

ABSTRACT

The increasing failure rate in High Performance Computing encourages the investigation of fault tolerance mechanisms to guarantee the execution of an application in spite of node faults. This paper presents an automatic and scalable fault tolerant model designed to be transparent for applications and for message passing libraries. The model consists of detecting failures in the communication socket caused by a faulty node. In those cases, the affected processes are recovered in a healthy node and the connections are reestablished without losing data. The Redundant Array of Distributed Independent Controllers architecture proposes a decentralized model for all the tasks required in a fault tolerance system: protection, detection, recovery and masking. Decentralized algorithms allow the application to scale, which is a key property for current HPC system. Three different rollback recovery protocols are defined and discussed with the aim of offering alternatives to reduce overhead when multicore systems are used. A prototype has been implemented to carry out an exhaustive experimental evaluation through Master/Worker and Single Program Multiple Data execution models. Multiple workloads and an increasing number of processes have been taken into account to compare the above mentioned protocols. The executions take place in two multicore Linux clusters with different socket communications libraries.

1. Introduction

The number of components in HPC systems is continuously increasing. On the one hand, there are more sockets per node to accomplish the demand of more performance, and, on the other hand, the deploy of system with more but less powerful components, allows to save power consumption. A growing number of components implies that the probability of failure increases as well. Although the Mean Time Between Failures (MTBF) of each component tend to be high, the system can fail frequently because it is composed by a higher number of them [34]. Moreover, reliability issues come up when chips are operated at significantly lower voltage [37].

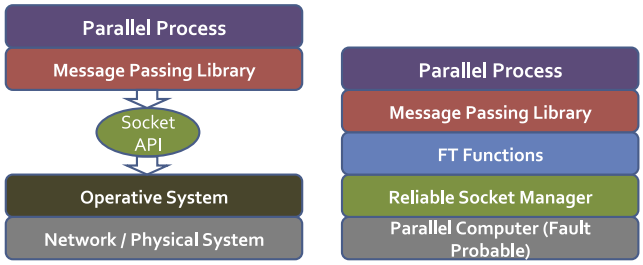
The remarkable increase in the failure rate of the systems [7,39] encourages the investigation of fault tolerance (FT) mechanisms. A node failure during the execution causes the loss of computation done until this point. The fault tolerance strategies have the aim to minimize fault's effects on applications to guarantee the execution in spite of node faults, in such cases in which the reliability to achieve a successfully end is very required.

Fault tolerance techniques based on rollback-recovery are highly recommended in the literature for message passing applications [14,13]. One of the most implemented approaches is based on coordinated checkpointing. It is a straightforward technique to recover the global state but it forces to roll back all the processes and the checkpoint coordination may slow down the application execution because of congestions on I/O [7,21]. However, current research is focusing on improving these scaling issues [35,32].

This research provides a transparent FT model, that is, it can be used without changing the parallel application and can be adopted at system level, independently of the message passing

* Corresponding author.
E-mail addresses: marcela.castro@uab.es (M. Castro-León), hugo.meyer@caos.uab.es (H. Meyer), dolores.rexachs@uab.es (D. Rexachs), emilio.luque@uab.es (E. Luque).

http://dx.doi.org/10.1016/j.jpdc.2015.08.005



(a) Parallel node. (b) Parallel computer.

Fig. 1. Fault tolerance software tiers.

library chosen. It is based on RADIC *Redundant Array of Distributed Independent Controllers* [38,12], an FT model architecture for message passing application, which works in a distributed and decentralized way during failure-free operations and recovery to allow the application to scale.

A complete FT model should provide a way to protect the state, detect faults as soon as possible and automatically recover a consistent state in order to finish the execution. Automatic recovery allows us to decrease the Mean Time to Repair (MTTR). The Availability, defined as the degree to which a system or component is operational and able to perform its designed function, is calculated as follows:

$$Availability = MTBF/(MTBF + MTTR). \quad (1)$$

Therefore, a reduction in the Mean Time To Repair (MTTR) implies a higher availability even though the MTBF of the system increases.

A message passing parallel application is able to resume and successfully end its execution in case of a node failure when it is provided for fault tolerant functionality. According to Gropp and Lusk [20], FT can be added at different layers with distinct degrees of transparency for the application.

Fig. 1(a) represents the software tiers in a parallel node. Usually, message passing libraries use the Socket Application Program Interface (API) to make inter-processes communications. The socket API is a de-facto standard for network programming in most used operative systems (OS) and provides means for interconnecting processes through the network. This API is widely used because of its simplicity, robustness and portability [2]. Fig. 1(b) aggregates the tiers in the parallel computer and it depicts the location of our proposal. Fault tolerance at system level can be used as a transparent service for the message passing library and for the application.

Our approach is located at system level and has two pillars. The first, which is called reliable socket manager (RSM), consists of replacing the socket communications for reliable connections which are able to detect and recover from network errors. Instead of returning a fatal error when a socket is closed due to a remote failure, the connection is reestablished again without losing data, by using the new IP address where the remote process has been migrated. Secondly, FT functions based on RADIC are devised at socket level in order to save the state of the processes and to recover only the affected ones when a node faults.

The actions related to saving the state, detecting the errors, recovering the processes state and keeping reliable communications are triggered when the processes use the socket API, which works at OS level. The default socket behavior is transformed into reliable connections with fault tolerance mechanisms by intercepting calls to socket API as a dynamic library. The mechanism is transparent not only for the application designer, but also for the system administrator because the failure is automatically detected and recovered, and the application is able to finish successfully without human intervention.

RADIC architecture has been applied as an FT model for two main reasons: firstly, because it has a completely distributed and decentralized behavior which allows the application to scale and, secondly, because it is based on an uncoordinated checkpoint with message logging. This rollback recovery protocol allow us to recover only the failed processes instead of recovering all of them [14]. Specifically, pessimistic receiver-based logging allows us to recover the failed process in an autonomous way. Furthermore, only the last checkpoint and the messages received from it have to be kept. However, that protocol is slower than other alternatives such as sender-based or causal algorithms.

In order to reduce the overhead of the log procedure, we propose using the semi-coordinated checkpoint [10]. Current HPC systems based on multicore processors tend to execute several parallel processes in each node of the system, usually following the strategy of one process by core. Semi-coordinated checkpoint consists of relating the processes running in a node because all of them are affected in case of node faults. The logging of messages among them is avoided and they are checkpointed coordinately. The receiver-based pessimistic log is applied for messages between processes in different nodes.

This paper provides a global vision of the system architecture presented in previous works [8–10] and the functionality of the following rollback recovery protocols are described and their impact in the performance is analyzed and compared.

- Fully uncoordinated checkpoint (NCO): It is the default RADIC protocol using uncoordinated checkpoint combined with receiver-based pessimistic message log.
- Blocking Semi-coordinated checkpoint (BSCO): It performs a blocking coordinated checkpoint [10] among processes in a node combined with receiver-based pessimistic message log for inter-node messages.
- Semi-coordinated checkpoint (SCO): It performs a non-blocking coordinated checkpoint among processes in a node combined with receiver-based pessimistic message log for inter-node messages.

Two popular distributed programming schemes which follow a well-known communication patterns, such as master–worker (M/W) and Single Program Multiple Data (SPMD), have been selected for the experimental evaluation. Both patterns allow us to analyze the overhead of the semi-coordinated alternative in the worst scenario, when no message is interchanged among processes on the same node, and, on the contrary, when there are internal communications.

Experiments using different input size are carried out to analyze and compare the FT overhead using distinct relations between the computation and the communication time needed for each task. In other sets of executions, the number of processes are increased as well in order to check whether the FT system affects the scalability of the application.

The content of this paper is organized as follows. In Section 2 we mention the related works. Section 3 defines the RADIC

architecture used as a model for our solution. Section 4 explains the devise of our approach at socket level and Section 5 describes the semi-coordinated rollback recovery protocols which can be selected as alternatives to reduce overhead. The experimental evaluation is presented in Section 6, and lastly, we state the conclusions and the future work in Section 7.

2. Related works

The first part of this section relates our work to current state of the art in fault tolerance for message passing application located at different system layers. The second part mentions other studies which change the default socket behavior in order to achieve resilience. The third part compares the semi-coordinated checkpoint algorithm with other similar approaches.

2.1. Fault tolerance for message passing applications

A message passing parallel application is able to resume and successfully end its execution in case of a node failure when it is provided for fault tolerant functionality. According to Gropp and Lusk [20], FT can be added at different layers with distinct degrees of transparency for the application.

The first is the application layer 1(a), which is one of the most developed approaches because FT is able to be executed efficiently. The best points where state changes are saved and where possible failures are more easily identified at this layer. Examples of projects in this category are, for instance, the methodology of Gropp and Lusk [20], the compiler to generate parallel checkpoint of Rodriguez et al. [36], or the techniques described by Florio [17]. Fault-Aware MPI, Hassani et al. research [23] and Scalable Multilevel Checkpointing System of Mohror et al. [32] are more recent research works also targeted for the application level. There are proposals which facilitate the development through the provision of libraries as well. However, they require a re-design of the application taking into account the strategy of protecting, detecting and recovery from node faults. It could be expensive in terms of time and effort in developing and testing. In addition to that, users are reluctant to add the FT in order to avoid errors caused by changes, and, it might be not possible to do it if the source code is not available.

In the second level, we found a variety of approaches which locate the FT algorithms in the communication library to be transparent for the application. Usually, FT is added as an extension of well-known MPI implementors like MPICH-V Project [5] or Open MPI [25]. RADIC was previously implemented using this kind of strategy [16]. Although the application is not changed, the source code is still needed to be compiled again with the modified communication library. Moreover, a specific FT strategy has to be adapted to different MPI implementations and releases. MPI standard still does not provide special functions to manage failures transparently and automatically, although there are current proposals to add new MPI primitives in order to handle node faults such as User Level Fault Mitigation of Bland et al. [3]. This proposal is currently being monitored by the MPI Forum [40].

Finally, solutions at system level are transparent for the application and do not have to use a specific message passing library. The application source is not required for adding FT. The message passing library does not need to be updated to support node failures. Our work fits in this last category. Projects in this group might be able to provide for the system administrator a way of defining a unique protection and recovery policy for the whole system, independently of which message passing library is in use for the hosted applications. An example using this last category is DMTCP [1], a checkpoint and restart tool for distributed applications which can also be used for MPI. However, it does not provide detection and automatic recovery without administration intervention as our work. Furthermore, RADIC provides uncoordinated checkpoint with message logging rollback recovery protocol but DMTCP, so far, only implements coordinated checkpoint, which is not advisable to scale the number of processes [6].

2.2. Fault tolerance at socket level

Our approach of FT is supported by the reliable socket manager (RSM), which transparently transforms the TCP sockets used by the application into reliable ones. The reliable sockets are able to recover from TCP crashes and eventually, to change the destination address without loss of data. Zandy et al. [41] developed a similar approach to building reliable TCP connections to keep a distributed application running despite network crashes. RSM goes further by adding control channel to preserve the integrity of the data interchanged by the application and it saves the data connection for modifying it before reconnection.

Ivaki et al. [27] proposed building a fault tolerance session above the socket layer which is able to recover the TCP session in case of crashes and includes a checkpoint facility for server side. RSM gives the same detection and reconnection service for both client and server sockets, because it is intended to be used for parallel applications instead of client and server ones.

2.3. Checkpoint schemes with less overhead

As a general rule, the more the messages are interchanged by the application, the more overhead is introduced by the message logging approach. In failure-free execution, such overhead is even worse when receiver-based logging approach is used [33]. Semi-coordinated checkpoint [10] relates the processes running in a node and they are checkpointed coordinately. Therefore, the messages interchanged between processes in a same node are not required to be logged, thus reducing the total log overhead.

This paper proposes a semi-coordinated checkpoint protocol which uses either blocking or non-blocking coordination procedures. Previous research works have presented this kind of strategy, but with three types of differences. Firstly, related to the criteria to group the processes, secondly, the type of coordination among processes or groups and, finally, the selected message logging protocol.

Bouteiller et al. [4] defined a correlated set coordination among processes executing on the same multicore node combined with sender-based pessimistic message logging. Sender-based protocols are faster than receiver-based ones during failure free operations but the latter is more efficient in recovery. When failure occurs in sender-based cases, processes that were not involved in the failure may need to re-send messages to restart processes or a centralized controller has to be included to do that. Receiver-based approaches are able to recover the state in an autonomous and faster way. Another difference is that the experimental evaluation is done using an adapted Open MPI transparent for the application while we use RADIC at socket level [9,8].

Luo et al. [30] proposed a combination of coordinated and uncoordinated checkpoint. It is targeted to grid environments. Those processes, which frequently communicate, are grouped together. For obtaining a consistent state, Communication Induced Checkpoint (CIC) combined with a pessimistic message logging is applied. CIC might not be scalable for highly coupled processes, since the number of forced checkpoints grows uncontrollably.

Group-based coordinated checkpoint is stated by Gao et al. [18]. The coordinated checkpoint is performed in several groups of processes with the aim of reducing the storage and network bottleneck. The checkpoint groups are formed in a way that the most frequent communication happens within groups. The rest of

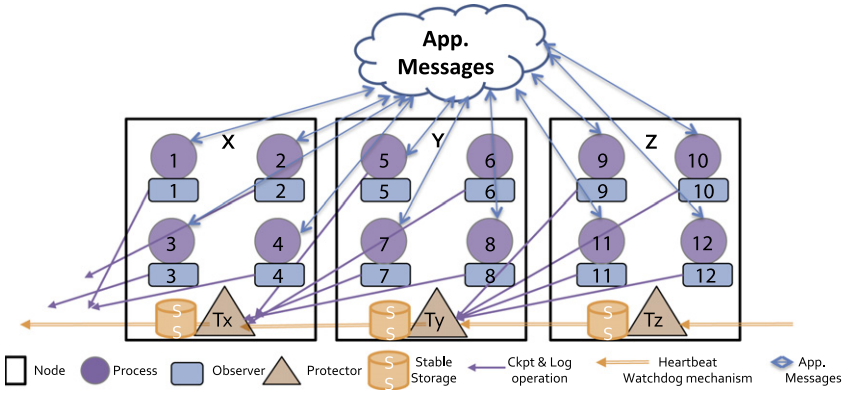


Fig. 2. RADIC diagram.

the processes wait for their checkpoint turn while they continue the execution unless there is no message to or from a process in checkpoint. This technique is suitable to reduce the failure-free coordinated checkpoint congestion but the recovery should include all checkpoint files. It is applied to MVAPICH2, then the approach is associated with a specific message passing library.

A combination of coordinated checkpoint with message log is proposed as well by Ho et al. [24], as a way to scale the most extended strategy of coordination of the whole processes. The criteria for grouping the processes is based on the communication behavior instead of using the failure correlation in a node. This level of knowledge implies an additional overhead for monitoring. The messages interchanged among groups are logged by the sender.

In order to reduce the overhead of coordination task and to allow to scale, Chandy et al. [11] proposed a method which is based on sending marks to other processes to know the global state. Several algorithms improve the mechanism by trying to reduce the number of processes involved in the coordination [28]. Our non-blocking coordination algorithm is similar to the mechanism adopted by Ho et al. [24] since the consistency among groups is guaranteed by the message log. However, in our system the coordination is done in each node independently from the others while in Ho's approach, the checkpoint initiator is a global process for all the groups.

3. RADIC architecture

RADIC [38,12] is a fault tolerance architecture which defines the protection, detection, recovery and error masking models to guarantee a successful message passing execution in case a system node suffers a failure.

RADIC provides current appealing properties such as transparency and decentralization. It is transparent in the sense that the application does not have to be modified at all. The components and functions used to tolerate failures are distributed and the decisions are decentralized, based on local information. Consequently, the fault tolerance functional phases, including recovery, do not need either central elements or collective operations. These properties allow the application to scale. Lastly, the failure detection and the recovery work automatically without needing administrator intervention.

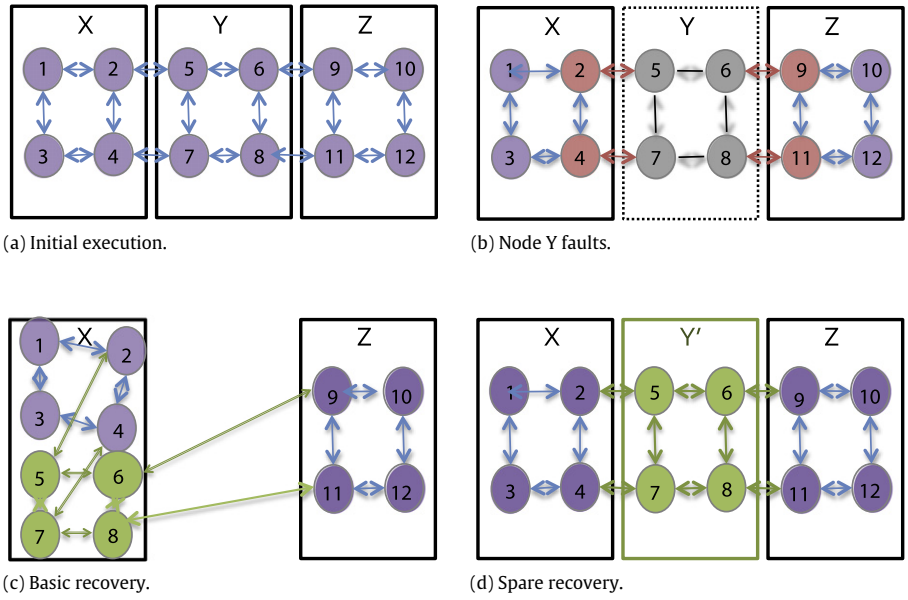
Protectors and observers are the two distributed software RADIC components which work together to carry out tasks of protection of the execution state of the processes, failure detection, processes recovering and lastly, masking the errors to the application processes.

There is a protector running in each node and one observer attached to every application process. A data structure named RadicTable is used to know the relation among nodes, observers and protectors. This structure is managed distributively and decentralized, updated on-demand by the RADIC components, using a deterministic algorithm which guarantees consistency among the nodes.

Fig. 2 represents an application running in fault-independent nodes using RADIC. We use i to represent each node N_i and protector T_i, and j for process P_j and for it observer O_j. The processes running on a node are assigned to a protector in another node, to save the state of the execution of their processes. For instance, the nodes could be numerated sequentially and the previous one is assigned in a circular way. Processes running on N_i send logs and checkpoint files to protector T_{i-1}, In case of failure, the sequence is broken but is reestablished by skipping that node. If there is a spare node to replace it, it will take the place in the sequence. The deterministic algorithm follows this procedure to update the RadicTable in each node.

The RADIC functional phases are explained as follow.

- **Protection model**: The execution state of the processes is saved during the failure-free execution. Observer O_j is in charge of protecting the state of the process P_j. By default, RADIC uses uncoordinated checkpoint combined with receiver-based pessimistic message logging which has the advantage of being decentralized in protection and recovery. The observer O_j has at least one protector assigned, which is located in a different node. The log messages and periodic checkpoints are sent to the protector. The number of assigned protectors is related to the grade of protection, which is the number of simultaneous supported failures. In this paper we assume that the grade is one. The protector T_{i-1} is in charge of saving the checkpoints and log messages.
- **Detection model**: The fault detection is carried out during the failure-free execution. A node faults when the processes running in other nodes are not able to communicate with the processes in it. Each protector T_i is in charge of detecting a possible failure of the neighbor node N_{i+1}. It uses heartbeat/ watchdog protocol. To reduce the latency of detection, determined by the configured interval of heartbeat, the observers warn to the corresponding protector when a communication error with one of the processes assigned to it is detected. For instance, in Fig. 2, if P_3 receives an error while it communicates with P_9, O_3 intercepts the error and notifies it to protector T_y which verifies if there is a fault in the node N_z.
- **Recovery model**: The protector T_i restarts the processes which were running in the failed node N_{i+1} using the data saved during protection time. The protector rolls back the failed processes restarting them from the last checkpoint. During the re-execution until the point of failure (roll-forward), the attached observer is in charge of delivering the logged message to restarted processes. The protector provides the saved message



(a) Initial execution.

(b) Node Y faults.

(c) Basic recovery.

(d) Spare recovery.

Fig. 3. FT at system level overview.

log to the observers. If the system has a spare node, the processes are restarted in it and the computational capacity is maintained. Otherwise, the processes are restarted in the same node N_i, which is likely to suffer a possible loss of performance, due to node N_i being likely to be overloaded after failure.

- **Error masking model**: The observer O_j masks the communication errors caused by a node failure by reconnecting P_j with the recovered processes in the new location. The protector is in charge of communicate to the affected observers, the new address where the processes have been recovered. The new location is updated in the RadicTable, and the following communications are done as usual. Another masking function is performed by the observer when it detects and discards duplicated messages coming from recovering processes.

4. Virtualization at the socket layer for fault tolerance

This section explains the fault tolerance model at socket level through the next subsections. First of all, a general overview of the proposal is described. Secondly, the socket API model is reviewed, since the FT actions are triggered when the processes use this API. Thirdly, the reliable manager, which is in charge of building the reliable sockets, is explained indicating how it works and for what it is useful. Finally, there is a subsection for describing each RADIC FT functional phases of protection, detection, recovery and error masking at socket level.

4.1. FT at system level overview

Fig. 3 provides an overview of how the processes and the communications are recovered after a node fault. An example of a parallel application, which follows a two-dimensional SPMD communication pattern, is represented in Fig. 3(a). Each process communicates with its neighbors. Fig. 3(b) shows how the processes 5, 6, 7, 8 in the faulty node N_y are fallen and therefore, the communication with its neighbors cause an error in processes 2, 4, 9, 11. Fig. 3(c) illustrates how the processes and the communications are recovered in the neighbor node. Finally, the last picture 3(d) outlines how the processes are restarted in a spare node and the connections are reestablished as well.

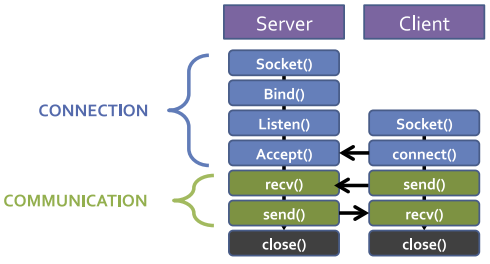


Fig. 4. Socket basic API model for server and client processes. Functions for connection establishment and communication.

4.2. Socket API model

Portable Operating System Interface (POSIX), the most extended platform in commodity Linux clusters, defines Socket [31] as *de facto* standard application program interface (API) to interchange data packages between two processes using transport level protocols like TCP or UDP [19].

Fig. 4 represents the most used and basic functions for server and client processes. There are some functions used to establish the communications and some others to communicate data.

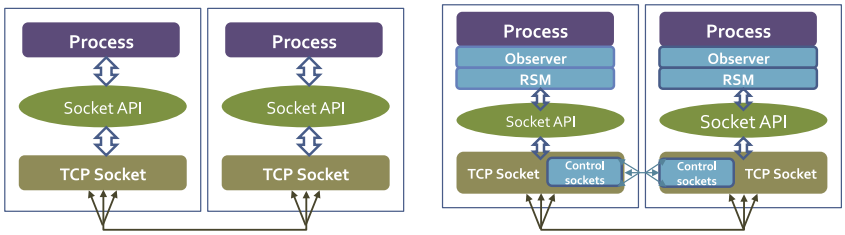
A communication is formed by a server socket opened with an accept() and with a client socket which performs a connect(). A socket identification is assigned, which is used to perform the data communication using basically send() and recv() operations.

When a node of a Linux cluster has a fault while a message passing application is running, the communications with the processes located in the rest of the nodes fail as well and in-transit data might be lost. A remote failure is treated by this API as a fatal error. Therefore, the connection is dropped and an error is raised.

Controlling socket errors caused by a failure of a remote peer prevents the propagation of them to the upper levels, which are the message passing communication library and application. Instead of using standard sockets, our solution proposes a mechanism to provide reliable sockets which are able to fail over from fatal errors.

4.3. Reliable socket manager

Reliable sockets are based on the OS standard sockets. The interception mechanism provided by POSIX systems is used in



(a) Default socket connections. (b) Reliable socket connections.

Fig. 5. Default and reliable socket connections.

order to add functionality but afterwards, the default socket API is called. The reliable socket manager (RSM) is in charge of building and maintaining the reliable sockets established by each process. RSM is integrated with the RADIC's observer as it is attached to each process. As they have the same API interface as default sockets, they are used transparently, namely, no change has to be made in the application in order to use reliable sockets.

Fig. 5(a) represents the default socket connection made by a process using the socket API. On the right, Fig. 5(b) shows the process with the RADIC observer attached to it using RSM to build reliable sockets.

RSM assumes that another RSM is located on the remote peer. One of them is the client, which performs a connect() API to establish a session and the other performs a listen() and an accept() for each connected client.

Once two RSM are connected, a socket control is created to convey acknowledge data of received messages and to interchange process identification, which is useful during the reconnection. This out-of-band channel also gives support to the message logging procedure which is further explained.

The socket control is created following the same sequence of the intercepted functions from the processes. Therefore, a client socket has a client control socket and the same for a server type. In such cases, the listen port is calculated taking the port of the process as a base plus a pre-configured displacement. There is a control socket for each connected socket.

These are the three states of the reliable sockets:

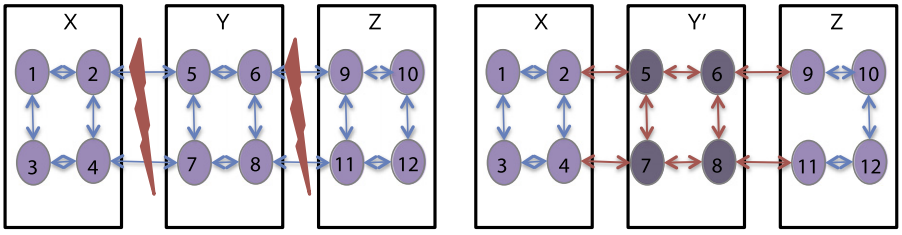
- *Creation*: This is the initial state from the declaration of the socket until it is connected using a *connect* or an *accept* operation for client and server sockets respectively. During this state, RSM creates the control socket and saves all the connection parameters in a data structure named Socketable, which is further detailed. As soon as the two peers are linked up, a first message is interchanged with the data to identify uniquely the remote RSM channel. This information is used by the server socket during reconnection to distinguish among several clients which could be running in the same node.
- *Connected*: This state is kept along the execution without failures, while the process sends and receives data. RSM detects socket errors by monitoring the returning result of the calling socket API. The recovery state is reached when the OS returns a fatal error indicating that the socket has become invalid.
- *Recovery*: When a socket fatal error is detected, RSM tries to reconnect with the remote process. If it is successful, the loss of connection would be likely to be caused by a remote checkpoint or by a temporary loss of network connection. Otherwise, RSM obtains the new IP address where the processes are recovered. Then, the connection is reestablished using the parameters saved during the creation state but changing the destination address. The API *connect* or *accept* is performed again in order to reconnect the processes. After that, the unacknowledged messages are resend to remote process.
- *Masking*: This state is reached after a communication failure. A new socket identification is assigned during reconnection. As the application is not aware of the change, the next socket operations called by the process after the failure are done using the original socket. RSM identifies as *virtual socket* the initial id known by the process and as *real socket* to the current id used to reach destination. After failure, RSM replaces the virtual socket with the real socket id before calling to the function socket to mask the change.

The intercepted socket's API parameters are saved in a local data structure named *Socketable* in order to repeat the same operation during recovery. There is a row for each socket connection of the process. The fields of the table are as follows:

Field	Description
Virtual socket	Socket id known by the process. It is obtained with *socket* or *accept* API for client or server socket respectively.
Real socket	Initially, it is equal to the virtual socket. It would change after a recovery (masking state).
Control socket	Id of the control socket.
Parameters	They are the ones used in the *socket* function (family, type and protocol).
Address	Remote IP and port assigned by the OS. Obtained after *bind, accept* or *connect* APIs.
Socket type	Client or server depending on whether the connection is established with *connect* or *accept*.
Socket sets	User socket configurations such as buffer size for instance, done with *setsockopt* API which are used to recreate the socket with the setting needed by the process.
Remote peer id	Remote process identification obtained during the first interchanged message. It is formed by the node id, the process identification (PID) and virtual socket. It is used during reconnection of a server socket to distinguish among several clients.

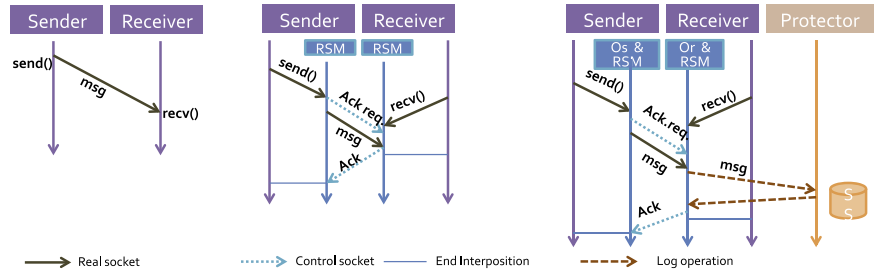
Special attention is required when a process loses several connections. A deadlock can occur while RSM is trying to reconnect a virtual socket. A possible scenario is graphed in Fig. 6 as an example. The processes affected by the failure have more than a connection to recover. A deadlock would appear if P5 tries to reconnect with P7, P7 tries to reconnect with P8, P8 to P6 and so on. Since all the processes are waiting for a process which is busy in other task, a deadlock occurs.

A deadlock would be avoided if the reconnection is carried out following a global order, being aware of which connections have fallen in each process. However, in our model, the observer prevents deadlocks without managing centralized information about



(a) Broken connections in a node fault. (b) Possible deadlock in socket reconnection.

Fig. 6. Deadlock risk during reconnection.



(a) Default socket message. (b) Reliable socket message. (c) Rec.based pessimistic log.

Fig. 7. Message interchange instrumentation for error detection and message logging. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the connections of the rest of the processes. The aim is to use the available local data managed by each process to keep a decentralized behavior. In order to do that, RSM assumes, after detecting a fatal error in a socket, that the rest of the connections with the same destination address are also affected by the failure. Then, each process would reconnect all the connections with the same target concurrently. With this assumption, following the previous example, P2 to P11 assign a slot of time to recover iteratively their failed connections, thus avoiding having a deadlock.

4.4. RADIC protection at socket level

The protection phase is referring to the tasks done for saving the state of the processes. According to RADIC model, the observers perform uncoordinated checkpoint combined with receiver-based pessimistic message logging.

Fig. 7 represents the message interchanged between two peers. Fig. 7(a) depicts a default message transmission using send and receive functions. The solid line represents the real socket between two processes used to transport the message. Fig. 7(b) shows how RSM interposes the *send*() function calling. The clear blue dotted lines depict the control socket. An acknowledge requirement and the message are sent through the control socket and real socket respectively. After that, it keeps waiting for an ack to assure the delivery of the message and to detect a possible fall. In the other peer, the *recv(X)* function is interposed and the *ack–req* and the message are read. The message is read completely as the length is indicated in the requirement. An acknowledgment is returned. The receiver would perform more than one read to finish the buffer. Lastly, Fig. 7(c) represents the procedure followed by two peer observers, which include RSM, to perform the receiver-based message log. In this case, the observer receiver O_r, sends the message to its neighbor node, which acts as stable storage. The dashed orange lines show the connection with the protector node.

The checkpoint to save the state of each process is performed by the observer using BLCR library [22]. The result is saved in a file which is sent to the protector. No coordination with other processes is needed since the message logging prevents the loss of in-transit and the generation of orphan messages. The checkpoint frequency is a configuration parameter provided by the user. There are several models to determine a suitable checkpoint interval [15].

The protector needs to keep the last checkpoint and the following received messages of each process running in the next node. This is the critical data needed to recover the processes until the point of failure, when a receiver-based pessimistic message log is used. As the nodes fault independently, the information is considered in a stable storage once it arrives safely to a different node. As long as there is enough memory, it is recommended that the protectors keep such critical data in memory in order to have a quick response during failure free execution and during recovery too.

4.5. RADIC detection at socket level

When a node faults, the current connections with the processes running in it are broken. Therefore, the first detection could be done by any of the affected processes. Fig. 8 depicts the broken connections.

A socket error could be detected while the observer is sending a message or while it is sending the log or the checkpoint to its neighbor. In such cases, the observer raises an error and asks for the state of the remote peer to its protector. The location of such process is obtained from the RadicTable. In the depicted example, observer 9 asks to T_x about the state of $Node_y$ after receiving a socket error.

Protector T_x is responsible for diagnosing the state of its protected $Node_y$, from which it has been receiving the critical data. It periodically receives a heartbeat message coming from T_y to be sure it is still working. Following Fig. 8 where $Node_y$ faults, T_x reaches the diagnostic procedure after detecting the fault by itself (heartbeat) or after receiving a question about the state of node $Node_y$ coming from any of the rest of the Observers or Protectors.

The diagnosis procedure is followed in order to know whether the node T_y has really failed or if the received error has been caused by a network outage or by a process checkpoint. Such a situation is treated as a Byzantine problem [29] and it is assumed that only one node could be faulty during the diagnosis.

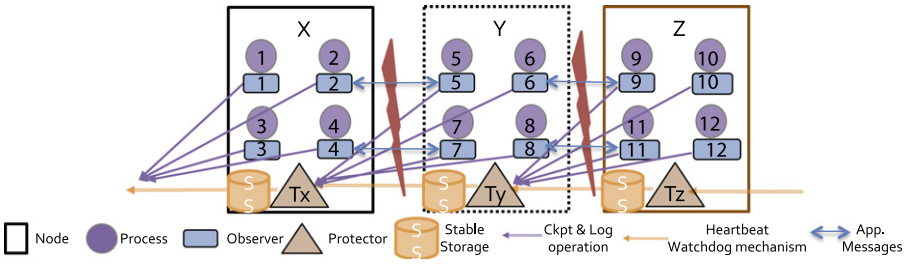
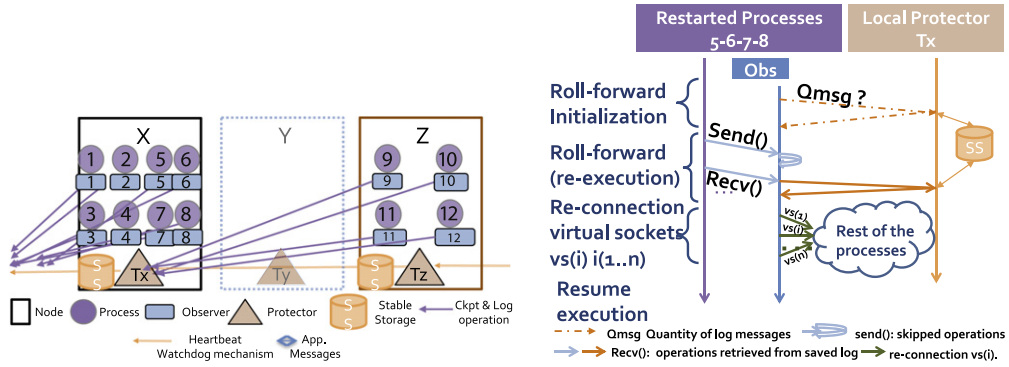


Fig. 8. RADIC detection model for $Node_y$ – T_x diagnoses – T_z confirms.



(a) RADIC recovery model. (b) Roll-forward procedure.

Fig. 9. Recovery and re-execution model.

As each node has two related neighbors, the protector contacts with the successor node T_z in order to determine the kind of failure and takes one the following actions:

- If there is no answer, it means that T_x cannot communicate with more than one node simultaneously. Since only one node could be faulty, the error is due to network problems. Therefore, T_x terminates with the processes running in its node and with itself, and the state will be recovered by protector T_{x-1}.
- If the successor T_z answers that the suspect T_y is fine, the protector assumes that it has a communication problem with its successor and does the same as described above.
- If the successor T_z confirms that the suspect T_y failed, the protector T_x assigns it as the new successor and the recovery state is reached.

There are three possible outcomes from the diagnostic procedure, which are used to answer to the observers stopped by the communication error.

1. Node N_y is working correctly, the protector is still receiving heartbeat from it. Therefore, the failure was caused by a temporary network delay and the lost communications will be recovered after some retries.
2. Node N_y is working correctly but at least one of its processes is performing a checkpoint. During checkpoint, the communications are released. Before beginning a checkpoint, the observer sends a message to its protector. The observers having a problem to communicate with it will recover the lost communications after the process finishes the checkpoint, since RSM manage such re-connections.
3. Node N_y has fallen down and the processes are being restarted using the last received checkpoint in another node. The new node's IP direction is sent to the affected observers. After receiving the new address, the observers use the RSM facility to change the destination address of the affected reliable sockets and they are connected again.

After receiving the diagnostic result, the observers continue on retrying to recover the lost connection using the same destination address (for answers 1-2) or changing it by the new one (answer 3).

4.6. RADIC recovery at socket level

Fig. 9(a) depicts how the system is reconfigured after a failure on the $Node_y$ when no spare node is available. The recovery is started by protector T_x after executing the diagnose procedure which confirms the fault of $Node_y$.

Observers hosted by $Node_z$ redirect their logs and checkpoints to protector T_x and T_z sends heartbeat to T_x as well. T_x restarts the failed processes in $Node_x$ using the last checkpoint.

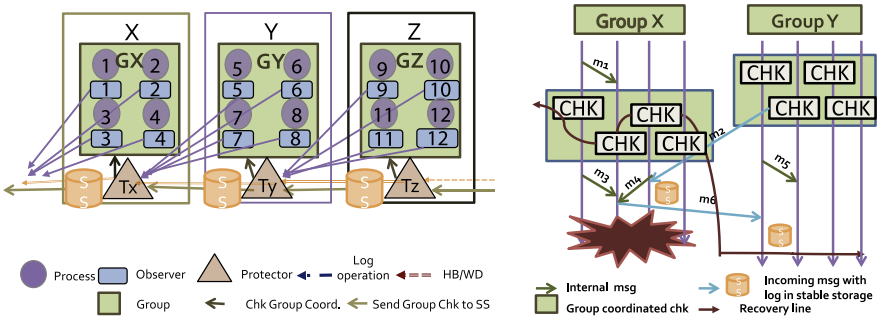
The observer of the restarted process follows the roll-forward procedure graphed in Fig. 9(b) and explained as follows. It is assumed that the process has n active connections which were saved by the RSM in the Socketable.

1. The quantity of messages logged (Qmsg) from the last checkpoint until the point of failure is queried to local protector.
2. During rolling forward, the sends are intercepted and skipped to avoid repetitions. Instead, the Qmsg received messages are intercepted and the content is retrieved from the local protector's stable storage.
3. Once the messages (Qmsg) are totally consumed, the n virtual sockets are reconnected.
4. The execution is resumed from the previous failed point.

4.7. Error masking model

These are the situations in which the observer masks the error caused by node faults to the processes:

- When a fatal socket error is detected, instead of raising an error, RSM reconnects to the same IP address or to a new one depending on the diagnose answer received from the remote peer's protector.



(a) Group of processes. (b) Semi-coordinated execution.

Fig. 10. Semi-coordinated checkpoint protocol. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

- When a process is restarted, the sockets are recreated again and the connections are reestablished in a transparent way for the upper level. The clients are able to locate the new address after receiving the diagnostic result from the protector and the client connections are replayed using the data saved in Socketable.
- When a process is restarted, after reconnection, it is possible to re-send a message. This situation happens when a failure occurs after sending a message and not after receiving one. As the RSM numerates the messages in the acknowledgment requirement, the receiver RSM is able to detect and discard repeated messages.

5. Semi-coordinated protocol for multicore systems

The rollback recovery protocol adopted by RADIC based on receiver-based pessimistic message logging has the advantage of being decentralized in protection and recovery, but it introduces more overhead than other algorithms during failure-free execution. An alternative rollback recovery algorithm called semi-coordinated checkpoint is proposed to reduce the overhead of message logging. Both blocking and non-blocking semi-coordination, which are named BSCO and SCO respectively, are presented and discussed in this section. The previous explained protocol based on non-coordinated checkpoint and receiver-based message log is named NCO.

The overhead is mainly caused by the sender, which has to wait for the ack to be assured that the message has been received and logged before continuing the execution, as it is shown in Fig. 7(c). The drawback of this protocol can become even worse when the application is running in a multicore system in which several processes are running in a node usually mapped one-by-core. Consequently, a sender and receiver might be in the same node. In such cases, the receiver-based protocol could introduce disproportional latencies. Moreover, processes are usually located together when they are tightly coupled by messages, therefore, they are even more penalized.

However, in case of a node failure, the processes running in it should rollback until a consistent point in a past before rolling forward until the point of failure. Therefore, there is natural coordination among processes running in a same node in case of failures.

Coordinated checkpoint among several processes is done to reach a consistent global state in order to have a recovery line. A process can be carrying out computation or communication (sending or receiving data). The coordination aims to avoid in transit messages which lead to an inconsistent recovery state. The processes must be computing or waiting for communication during checkpoint. Message logging protocols allow us to avoid coordination and to restart only the failed processes. By logging the messages during execution and replaying when they are rolling forward, the local state is recovered. [14].

Semi-coordinated checkpoint combines the two techniques. On the one hand, the processes running in the same node form a group and the state is saved coordinately, being assured that they are not carrying out communications. On the other hand, as the processes in a group can communicate with a process in other group, a message logging is done in order to guarantee building a consistent recovery line among groups.

Fig. 10(a) depicts the group of processes in each node. The protector running in the same node (local) is in charge of coordinating the checkpoint.

The semi-coordinated protocol BSCO changes the protection model as follows:

- The local protector coordinates the checkpoint among processes in a group. Periodically, a message is sent to them indicating that a checkpoint should be initiated. After receiving such message, the observer stops the communications and replies to the protector, indicating that it is ready. Then, the checkpoints are performed by the local protector and they are sent them to the protector located in other node. In Fig. 10(b), the light green rectangles represent the coordinated checkpoint performed on the groups G_x and G_y.
- The messages interchanged between processes in the same group, which are named *internal*, do not have to be logged since they would be replayed again if the node fails. Examples of such messages are $m1$, $m3$, $m4$ $m5$ shown in Fig. 10(b). However, the order of events is logged to avoid generating orphan messages.
- The messages interchanged between processes in different groups such as $m2$ and $m6$ are named *incoming* or *outgoing* depending on whether they are arriving or going outside the group. The log of *incoming* messages is carried out by the receiver, following the same procedure explained previously in Section 4.4 and represented in Fig. 7(c).

In Fig. 10(b), the red line represents the points of recovery when the group G_x is affected by a node fault. Even though G_y is not failed, the processes, having active connections with failed process of G_x, resume their execution after reconnecting with restarting processes. On the contrary, processes with no connections with failed groups, are able to continue the execution and are not affected by the failure. The recovery model of BSCO is altered by this new protocol as well as the following.

- The protector restarts the failed processes using the last checkpoint files.
- The failed processes roll-forward together until the point of failure. As the incoming messages are not received again, the log is used (m2 in our example). On the contrary, *internal* messages are replayed.

Although the semi-coordinated protocol avoids logging internal messages, it introduces a coordination task which could make the

total execution time longer than the previous one, using uncoordinated checkpoint combined with receiver-based pessimistic message log (NCO). When only a few or none of the internal messages are interchanged, it is possible that semi-coordinated checkpoint does not manage to reduce the execution time. Moreover, when the coordination task is in the critical path, which means that it does not overlap with other computational tasks, our proposal might be longer.

5.1. Reducing the coordination time overhead

The blocking checkpoint coordination performed by the local protector in BSCO has the following steps.

1. When it is time to checkpoint, according to the configured interval, a checkpoint indicator message is sent to all the observers in the group.
2. Each observer stops at the end or at the beginning of the next communication operation to guarantee that there is no in-transit message.
3. The protector launches the checkpoint to all the processes in the node.
4. The protector indicates to the observers that they can continue the normal operation and the checkpoint files are sent to the protector located in the other node (stable storage).

The overhead is mainly caused by the barrier placed in the third step of the procedure. The processes are forced to wait until all of them are ready to be checkpointed.

There are several proposals in the literature to reduce the coordination overhead which were discussed previously in related works. In this work, we propose a non-blocking mode named SCO that reduces the coordination time by using a receiver-based logging for internal message only during the checkpoint time, in which the processes in a group are performing checkpoint. Therefore, in-transit messages are logged.

In more detail, the protector, performs a checkpoint of each process immediately after it says it is ready without waiting for the rest of the group processes to finish their tasks. After checkpoint, each process is able to continue but all the received messages are logged, even the internal ones, until receiving a message from the local protector indicating that all the processes have finished their checkpoints. In that way, no in-transit messages are lost and the wait time is reduced.

The recovery procedure in SCO also has to be adapted to this change too by adding a slight complexity and overhead. During restarting, the internal required messages look firstly in the log. If they are not there, it means that they are going to be replayed by other restarting processes.

In both protocols BSCO and SCO, the protector located in another node considers that a semi-coordinated checkpoint is successfully ended when all the checkpoint files corresponding with group members are received. Meanwhile, the files are saved in a temporary state. If the node fails before having all the checkpoint files, the previous complete checkpoint would be used to recover the group state.

6. Experimental evaluation

We have developed a prototype of RADIC at socket level. The observer is a dynamic library which builds reliable connections by intercepting socket functions and it performs the procedures of uncoordinated and semi-coordinated checkpoint using BLCR library [22]. The protector is a resident permanent service which interchanges heartbeats with its neighbors, as well as receiving and storing checkpoints and message log coming from the next node. It diagnoses communication errors, it restarts failed processes and re-establishes the detection order in case the next node fails.

There are three objectives in the experimental evaluation. Firstly, to accomplish the functional evaluation of the RADIC model at socket level using the default rollback recovery protocol (NCO). Secondly, we want to measure and compare the overhead of the FT system using the alternative protocols of fully uncoordinated checkpoint (NCO), Blocking Semi-coordinated checkpoint (BSCO) and Semi-coordinated checkpoint (SCO). Lastly, we execute the selected applications using an ascending number of processes from 24 to 64 using the same workload to check if the FT protection model leaves the application to scale.

Linux clusters. The experiments have been executed in these environments:

- *Cluster A*: It is formed by 4 dual-core nodes Intel® Core™ i5-650 Processor 6 GB RAM and Network Gigabit Ethernet. The OS is Ubuntu 10.04 Kernel 2.6.32-43-server. The nodes are named from N_0 to N_3.
- *Cluster B*: It is a Dell PowerEdge M600 with 8 nodes, each one with 2 quad-core Intel® Xeon® E5430 running at 2.66 GHz, provided with 16 GB of main memory and a dual embedded Broadcom® NetXtreme IITM 5708 Gigabit Ethernet. The operating system is Red Hat 4.1.1–52.

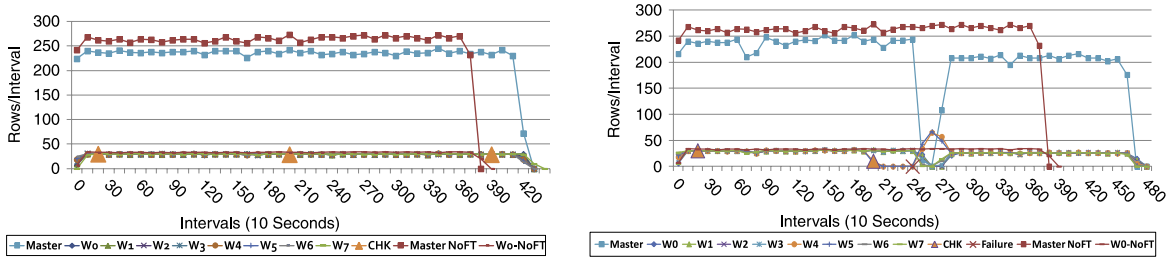
Applications patterns. We use two socket-based message passing parallel applications, which are the two most used patterns in HPC. First, a sum of float matrices with a Master/Worker (M/W) communication pattern, and secondly, a heat-transfer simulation which has SPMD behavior. Both patterns allow us to test the recovery and error masking model using a variable number of clients and server sockets per process. Moreover, a suitable comparison among proposed rollback recovery protocol is possible as well. On the one hand, M/W model does not have internal communications and it is expected to be affected by the overhead introduced by the coordination task. On the other hand, SPMD processes communicate among them and they have more sockets opened to test the deadlock avoidance in reconnection.

Sum of Matrices (M/W) was executed using a master and eight workers, named from W_0 to W_7, two by node. The master, located in N_0, generates a square float matrix of 10 000 × 10 000 and iteratively sends the rows to sum (110 kb) to the workers and it receives and accumulates the result (11 bytes).

Heat Transfer application (SPMD), represents a cell-to-cell communication achieved via exchange of temperature and thermal fluxes between neighboring cells. It was executed with eight processes, two by node. A square float matrix of 1000 × 1000 is uniformly divided among the processes. Each one makes a calculation during 100 000 iterations. For every 20 iterations, the processes interchange the edge rows with its neighbors processes. The package size is 4k.

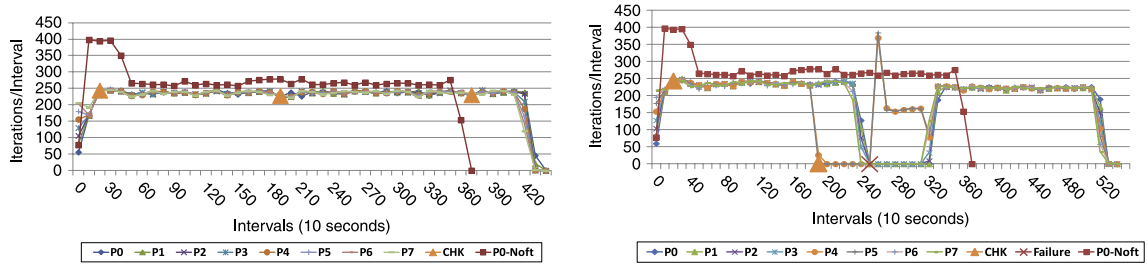
Type of executions. There are three types of executions used to evaluate, measure and compare the overhead of the FT system.

1. *No FT*: The applications are executed without FT. It is used as a reference to analyze the behavior changes and the overheads introduced by RADIC at system level.
2. *Protection*: RADIC is used for a failure-free execution. The protection phase is tested using the protocols NCO, BSCO and SCO. As the checkpoints close the sockets connection causing socket failures to the other processes, the checkpoints are only performed in the processes in one node to avoid overheads in diagnostics and reconnection to facilitate the overhead protection comparatives. The selected node for checkpoint is N_2 with an interval of 180 s. A initial checkpoint is triggered in the 20th second, considering that the application initialization is over after that.



(a) Protection execution. (b) Recovery execution.

Fig. 11. Master/Worker executions using fully uncoordinated checkpoint (NCO). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



(a) Protection execution. (b) Recovery execution.

Fig. 12. Heat transfer executions using fully uncoordinated checkpoint (NCO). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3. *Recovery*: RADIC is used and a failure is injected in N_2 40 s after the second checkpoint. As no spare node is available, the failed processes are recovered in node N_1. Consequently, after failure, this node supports four processes. As we are using dual-core nodes, we can observe the loss of performance.

The experiments are represented with a throughput diagram graph which shows the number of finished tasks by intervals of 10 s. The tasks are measured by rows processed in M/W and by iterations in SPMD.

Experiments using fully uncoordinated checkpoint (NCO). Fig. 11 represents a selected execution of M/W in protection and recovery using NCO. The red line represents the rows processed by 10 s without fault tolerance (No FT). In both figures are slightly superior because a higher throughput is achieved. Checkpoints and message logging slow down the performance and the execution time is longer, as can be observed on axis X.

In Fig. 11(b) it is observed how the throughput falls to 0 from the last checkpoint up to the failure as a consequence of the loss of tasks during rollback of failed processes. However, the tasks by seconds climb during the period of re-execution because the needed messages are already available in the log. The executions have less throughput after failure because the node is overloaded with 4 processes. The recovery execution time is prolonged in 58 s in respect of the failure free execution. This time is determined mostly by the lost time between the last checkpoint and the failure (40 s) and the rest, by the re-execution time. The performance slows down after the failure.

Fig. 12 represents the SPMD processes of the heat transfer application. The red line depicts the No FT execution, which acts as a baseline to observe how the throughput of the rest of executions slows down because of the checkpoint and logging. The added execution time is observed on axis X. In these experiments, the impact is mainly due to the log because the checkpoints are short as the processes do not use too much memory. They last 1 s and their size is less than 1.5 MB.

The added execution time in recovery of SPMD represented in Fig. 12(b), of almost 100 s, is determined by the time 40 s between the last checkpoint and the failure, by the re-execution time (70 s) and, finally, although the host node N_1 is overloaded with 4 processes, instead of resulting in delaying the execution even more, it is accelerated by the internal communications of 4 processes running in the same node.

Measuring and comparing rollback recovery protocols. We execute both applications in protection and recovery using the same checkpoint interval and failure injection to measure and compare different rollback recovery protocols such as NCO, BSCO and SCO.

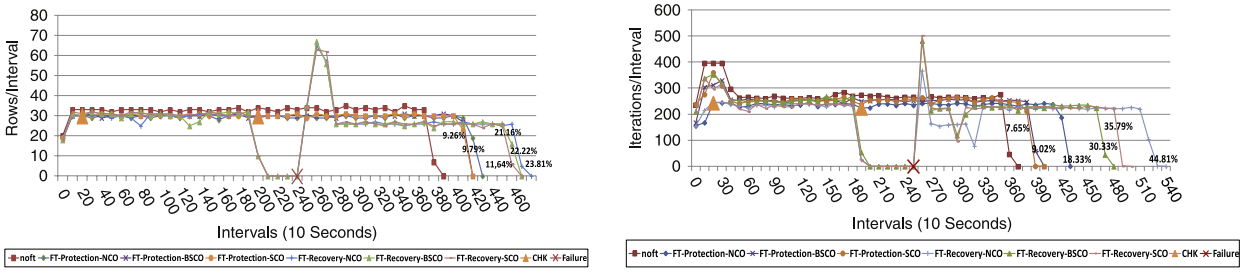
Fig. 13 represents the throughput of one of the two processes running on node N_2 in the distinct executions of No FT, NCO, BSCO and SCO. M/W is shown in Fig. 13(a) and SPMD in Fig. 13(b).

The higher the line is, the more throughput is achieved and the shorter the execution time is. As expected, Non-blocking semi-coordination (SCO) in protection has a better behavior both in M/W and SPMD, adding 9.26% and 7.65% of overhead to NoFT respectively. In SPMD, semi-coordinated proposals are much better than NCO due to the internal messages not being logged. In M/W, the differences are not pronounced to the fact that only the node hosting the master is shared with two workers. Consequently, the time saved for avoiding log in this node is compensated with the coordination time in all nodes when semi-coordinated is used.

NCO offers the worst time in protection and recovery adding 11.64% 23.81% in M/W and 18.33% and 44.81% in SPMD. As the amount of messages increases, so does the overhead. Moreover, NCO needs much more memory space to save the logs than semi-coordinated protocols BSCO and SCO, which are latter analyzed.

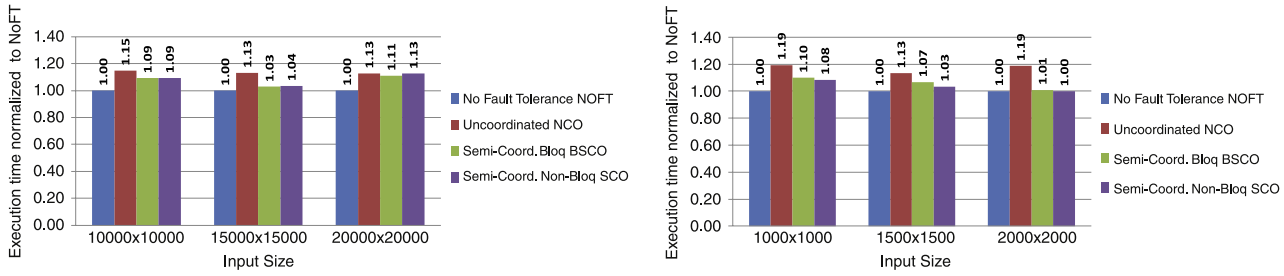
In SPMD program, BSCO is able to finish earlier than SCO. The coordination of the checkpoint might help to synchronize processes avoiding congestions, resulting in it being better for these kinds of patterns.

In order to analyze how the overhead is affected by the size of the input, we compare the average of at least ten execution times of M/W using sizes of square float matrices of 10 000, 15 000 and 20 000 and SPMD with float matrices of 1000, 1500 and 2000. The standard deviation for all the cases is less than 6%. As the input size increases, the communication to computation ratio varies and so does the impact of each protocol on the final time execution.



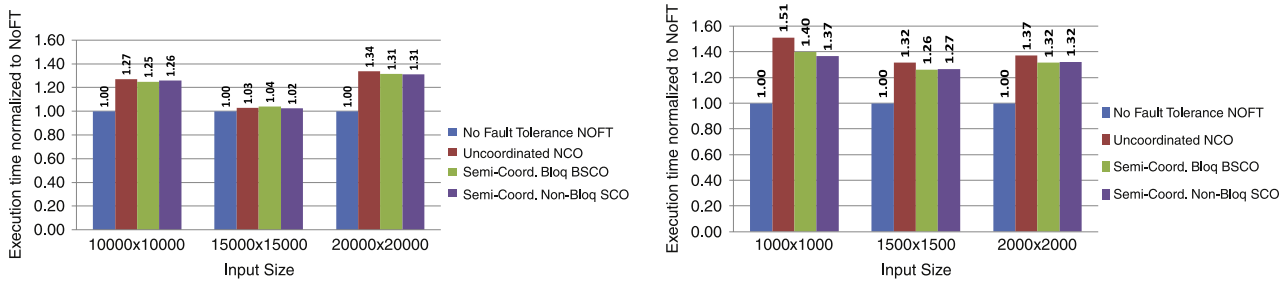
(a) Master/Worker. (b) SPMD.

Fig. 13. Comparative analysis of NCO, BSCO and SCO in recovery.



(a) Master/Worker. (b) SPMD.

Fig. 14. Time execution overhead of NCO, BSCO and SCO in protection varying the input size.



(a) Master/Worker. (b) SPMD.

Fig. 15. Time execution overhead of NCO, BSCO and SCO in recovery varying the input size.

In Fig. 14 shows how in failure-free operations, the NCO overhead in both patterns is higher or equal than other proposals for any workload. In MW, Fig. 14(a), NCO improves slightly as the size enlarges. BSCO and SCO makes the execution faster than NCO because in the first node the communications between the master and workers is avoided. When 15 000 square matrix is used, the communication and the communication is better overlapped, reaching the best performance.

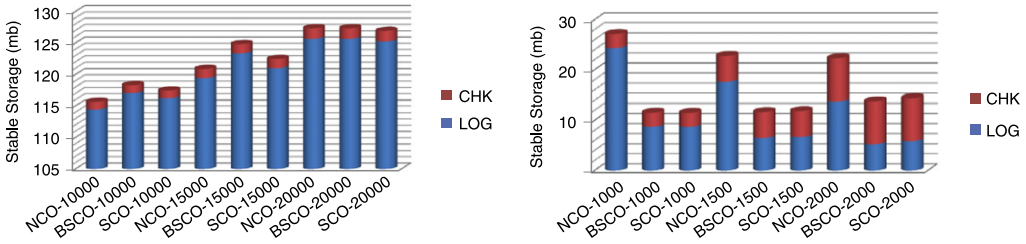
In SPMD, Fig. 14(b), NCO takes less time when 1500 square matrix size is used. The communication to computation ratio is more equilibrated in this case, achieving a better overlapping. However, in the semi-coordinated options, the longer the messages are, the less the overhead is as a consequence of the decreasing of the logs. BSCO and SCO reduce the overhead even more in heat transfer, as it takes more profit from reducing the task log than M/W, which only reduces the log of the master node.

Fig. 15 represents the overhead relation in execution time in recovery. The analysis is carried out taking into account that in all the cases, the rolling back is of 40 s. Instead, the recovery time depends on the amount and size of the lost messages for reprocessing and their availability.

In M/W 15(a), there are slight differences. BSCO shows the higher times when the messages take longer because it is much quicker to look for messages in log than wait for them, and moreover, a blocking coordination time is added. BSCO and SCO are better than NCO in SPMD 15(b). Again, it is shown how the coordination of BSCO can help SPMD to synchronize the processes before than non-blocking SCO and it manages to be faster in some executions. However, there are no higher differences since the overhead is highly determined by the rollback and by the re-execution time.

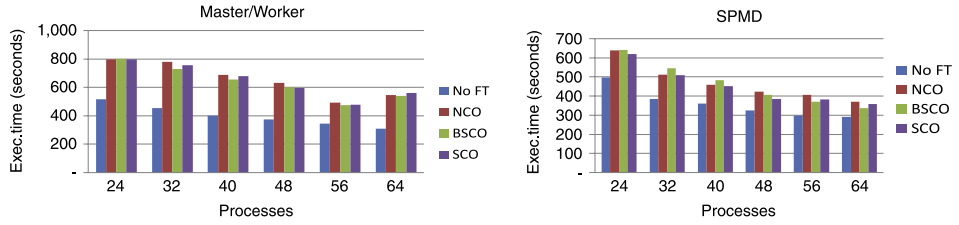
Furthermore, we compare the stable storage space used by the protocols for both patterns during execution using different input sizes. As we are using pessimistic protocols, only the last checkpoint plus the messages received after it are required for keeping in stable storage. In Fig. 16, the average stable storage size registered in node N_2 is represented. As was expected, there is no difference for M/W in Fig. 16(a) since all the messages are saved. Instead, in SPMD depicted in Fig. 16(b), the space needed in semi-coordinated protocol is highly reduced.

Lastly, we conduct an experiment in order to assess the scalability of the FT Protection model. We execute in cluster B, M/W and SPMD applications using square matrices of 20 000 × 20 000 and 2000 respectively. The number of processes vary from 24 to 64 using the same workload to check the weak scalability. The mapping of the processes to the cores follows a fill-up policy in all the executions. As each node has 8 cores, we start using 3 nodes as it is the minimum number of nodes to use RADIC. The protector is not assigned to any core in all the experiments. It is expected that the decentralized model allows the application to scale.



(a) Master/Worker. (b) SPMD.

Fig. 16. Stable storage usage of NCO, BSCO and SCO varying the input size.



(a) Master/Worker. (b) SPMD.

Fig. 17. Time execution comparison scaling the number of processes.

As is shown in Fig. 17(a), the performance of adding FT in Master/Worker is proportionally, excepting the No FT execution, with 64 processes in use decreases. It is caused by some inefficiencies in out implementation related to the use of normal vectors instead of hashed. However, in Fig. 17(b), the tendency of scaling follows the same as the No FT execution. SCO shows slightly better results than BSCO in SPMD, but up to 48 processes. As was explained before, the coordination of BSCO can help the SPMD application to synchronize the processes before the non-blocking SCO, then achieving faster executions.

7. Conclusions and future work

In this paper, we have presented a fault tolerance model which allows a message passing application to end successfully even when node failures occur. It is transparent for the application and it is able to detect node faults and recovering the processes automatically. As it is based on RADIC, it is distributed and has a decentralized behavior.

The model is located at system level, specifically at socket level. The application source is not required for adding FT. The message passing library does not need to be updated to support node failures, nor reconfigured or compiled again. The data and the procedures required to achieve a transparent and automatic failover mechanism at system level without losing in-transit packages are exposed through the reliable socket manager.

A blocking (BSCO) and non-blocking (SCO) semi-coordinated checkpoint protocols have been exposed as alternatives of the fully uncoordinated checkpoint (NCO) proposed by RADIC.

The experimental evaluation is carried out using socket message applications, which follow very well-known patterns in the scientific field, such as M/W and SPMD. It has been shown that the FT model is successfully applicable to both cases.

This paper demonstrates the feasibility of supplying FT functionality to message passing applications independently of the message passing library chosen, provided that it uses socket API to establish inter-processes communications. This feature would allow to avoid having to upgrade applications or message passing library without FT mechanisms, thus reducing the development and testing cost.

The results obtained from comparisons using different workloads, number of processes and clusters show that both SCO and BSCO are good alternatives for reducing time execution overhead when multiple processes with internal communications are executed in multicore clusters.

We are working on a set of experiments to demonstrate that we can use this middleware to fault tolerance applications, using either MPICH or Open MPI, the most popular MPI libraries. Additional socket functions have to be considered.

The model could be extended to other networks like Infiband [26] which provides application program interfaces that can be treated using socket API.

Acknowledgments

This research has been supported by 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] J. Ansel, K. Aryay, G. Coopermany, Dmtcp: Transparent checkpointing for cluster computations and the desktop, in: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, IEEE Computer Society, Washington, DC, USA, 2009, pp. 1–12.
[2] S. Atchley, D. Dillow, G. Shipman, P. Geoffray, J.M. Squyresz, G. Bosilcax, R. Minnich, The common communication interface (CCI), in: Proceedings—Symposium on the High Performance Interconnects, Hot Interconnects, Cci, 2011 pp. 51–60.
[3] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, J. Dongarra, Post-failure recovery of MPI communication capability: Design and rationale, Int. J. High Perform. Comput. Appl. 27 (3) (2013) 244–254.
[4] A. Bouteiller, T. Herault, G. Bosilca, J.J. Dongarra, Correlated set coordination in fault tolerant message logging protocols for many-core clusters, Concurr. Comput.: Pract. Exper. 25 (4) (2013) 572–585.
[5] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, F. Cappello, Mpich-v project: A multiprotocol automatic fault-tolerant mpi, Int. J. High Perform. Comput. Appl. 20 (3) (2006) 319–333.
[6] F. Cappello, Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities, Int. J. High Perform. Comput. Appl. 23 (3) (2009) 212–226.
[7] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, Toward exascale resilience: 2014 Update 2. The exascale resilience problem, 1 (1) (2014) 5–28.
[8] M. Castro, D. Rexachs, E. Luque, Radic-based message passing fault tolerance system, in: ADVCOMP, The Sixth International Conference on Advanced Engineering Computing and Applications in Sciences, 2012, pp. 59–64.
[9] M. Castro, D. Rexachs, E. Luque, Transparent fault tolerance middleware at user level, in: HPCS'12, 2012, pp. 566–572.

[10] M. Castro, D. Rexachs, E. Luque, Adding semi-coordinated checkpoint to radic in multicore clusters, in: PDPTA, 2013, pp. 545–551.
[11] K.M. Chandy, L. Lamport, Distributed snapshots: Determining global states of distributed systems, ACM Trans. Comput. Syst. 3 (1) (1985) 63–75.
[12] A. Duarte, D. Rexachs, E. Luque, Increasing the cluster availability using radic, in: Cluster Computing, IEEE International Conference, 2006, pp. 1–8.
[13] I.P. Egwutuoha, D. Levy, B. Selic, S. Chen, A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems, J. Supercomput. 65 (3) (2013) 1302–1326.
[14] E.N. Elnozahy, L. Alvisi, Y.-M. Wang, D.B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Comput. Surv. 34 (3) (2002) 375–408.
[15] L. Fialho, D. Rexachs, E. Luque, What is missing in current checkpoint interval models?, 2012 IEEE 32nd International Conference on Distributed Computing Systems, 2011, pp. 322–332.
[16] L. Fialho, G. Santos, A. Duarte, D. Rexachs, E. Luque, Challenges and issues of the integration of RADIC into open MPI, in: European PVM/MPI Users' Group Meeting, 2009, pp. 73–83.
[17] V.D. Florio, Application-Layer Fault-Tolerance Protocols, IGI Global, 2009.
[18] Q. Gao, W. Huang, M.J. Koop, D.K. Panda, Group-based coordinated checkpointing for MPI: A case study on infiniband, in: International Conference on Parallel Processing, 2007, ICPP 2007, 2007, pp. 47–47.
[19] R. Gordon, The TCP/IP guide: A comprehensive, illustrated Internet protocols reference, Lib. J. 131 (1) (2006) 146–146.
[20] W. Gropp, E. Lusk, Fault tolerance in MPI programs, Int. J. High Perform. Comput. Appl. 18 (3) (2004) 363–372.
[21] A. Guermouche, T. Ropars, E. Brunet, M. Snir, F. Cappello, Uncoordinated checkpointing without domino effect for send-deterministic MPI applications, in: Parallel Distributed Processing Symposium, IPDPS, 2011 IEEE International, 2011, pp. 989–1000.
[22] P.H. Hargrove, J.C. Duell, Berkeley lab checkpoint/restart (BLCR) for Linux clusters, J. Phys. Conf. Ser. 46 (2006) 494–499.
[23] A. Hassani, A. Skjellum, R. Brightwell, Design and evaluation of FA-MPI, a transactional resilience scheme for non-blocking MPI, in: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2014, pp. 750–755.
[24] J.C.Y. Ho, C.-L. Wang, F.C.M. Lau, Scalable group-based checkpoint/restart for large-scale message-passing systems, in: International Symposium on Parallel and Distributed Processing, 2008, IPDPS 2008, IEEE 2008, pp. 1–12.
[25] J. Hursey, J. Squyres, T. Mattox, A. Lumsdaine, The design and implementation of checkpoint/restart process fault tolerance for open MPI, in: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, 2007, pp. 1–8.
[26] InfiniBand®Trade Association: Home. URL: http://infinibandta.org/.
[27] N. Ivaki, S. Boychenko, F. Araujo, A fault-tolerant session layer with reliable one-way messaging and server migration facility, in: 2014 IEEE 3rd Symposium on Network Cloud Computing and Applications, NCCA, 2014, pp. 75–82.
[28] R. Koo, S. Toueg, Checkpointing and rollback-recovery for distributed systems, IEEE Trans. Softw. Eng. 13 (1) (1987) 23–31.
[29] L. Lamport, R. Shostak, M. Pease, The byzantine generals problem, ACM Trans. Program. Lang. Syst. 4 (3) (1982) 382–401.
[30] Y. Luo, D. Manivannan, Hope: A hybrid optimistic checkpointing and selective pessimistic message logging protocol for large scale distributed systems, Future Gener. Comput. Syst. 28 (8) (2012) 1217–1235.
[31] M.K. McKusick, K. Bostic, M.J. Karels, J.S. Quarterman, The Design and Implementation of the 4.4BSD Operating System, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.
[32] K. Mohror, A. Moody, G. Bronevetsky, B.R. De Supinski, Detailed modeling and evaluation of a scalable multilevel checkpointing system, IEEE Trans. Parallel Distrib. Syst. 25 (9) (2014) 2255–2263.
[33] S. Rao, L. Alvisi, H. Vin, The cost of recovery in message logging protocols, IEEE Trans. Knowl. Data Eng. 12 (2) (2000) 160–173.
[34] D.A. Reed, C. da Lu, C.L. Mendes, Reliability challenges in large systems, Future Gener. Comput. Syst. 22 (3) (2006) 293–302.
[35] R. Riesen, K. Ferreira, D. Da Silva, P. Lemarinier, D. Arnold, P.G. Bridges, Alleviating scalability issues of checkpointing protocols, in: 2012 International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2012, pp. 1–11.
[36] G. Rodríguez, M.J. Martín, P. González, J. Touriño, R. Doallo, Cppc: a compiler-assisted tool for portable checkpointing of message-passing applications, Concurr. Comput.: Pract. Exper. 22 (6) (2010) 749–766.
[37] T. Rosing, K. Mihic, G. De Micheli, Power and reliability management of SoCs, IEEE Trans. Very large Scale Integr. (VLSI) Syst. 15 (4) (2007) 391–403.
[38] G. Santos, A. Duarte, D. Rexachs, E. Luque, Providing non-stop service for message-passing based parallel applications with radic, in: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), in: LNCS, vol. 5168, 2008, pp. 58–67.
[39] B. Schroeder, G.A. Gibson, A large-scale study of failures in high-performance computing systems, IEEE Trans. Dependable Secure Comput. 7 (4) (2010) 337–350.
[40] User level failure mitigation — fault tolerance research hub, URL http://fault-tolerance.org/ulfm/.
[41] V.C. Zandy, B.P. Miller, Reliable network connections, in: Proceedings of the 8th Annual International Conference on Mobile Computing and Networking, MobiCom'02, ACM, New York, NY, USA, 2002, pp. 95–106.

Marcela Castro-León is an Associate Researcher of the group of High Performance Computing for Efficient Application and Simulations (HPC4EAS) at University Autonoma. She is an Associate Professor at the Tomás Cerdá Computer Science School. The research interests include fault tolerance in parallel computers, service oriented Architecture, service security, modeling and simulation.

Hugo Meyer earned his Ph.D. degree in High Performance Computing from the University Autonoma of Barcelona (Spain), where he worked on Fault Tolerance and Performance Analysis in parallel and distributed systems. Currently, he is working as a researcher in the Barcelona Supercomputing Center researching on Performance Evaluation and Simulation of optical devices in HPC and Data Center applications.

Dolores Rexachs is an Associate Professor at the Computer Architecture and Operating System Department at University Autonoma of Barcelona (UAB), Spain. She has been the supervisor of 7 Ph.D. thesis and has been invited lecturer in Universities of Argentina, Brazil, Chile and Paraguay. The research interests include parallel computer architecture, parallel I/O subsystem, fault tolerance in parallel computers, tools to evaluate, predict, and improve the performance in parallel computers. She has coauthored more than 50 full-reviewed technical papers in journals and conference proceedings.

Emilio Luque is a Professor at the Computer Architecture and Operating System Department at University Autonoma of Barcelona, Spain. Invited lecturer at universities in the USA, South America, Europe and Asia, key note speaker in several conferences and leader in several research projects funded by the European Union (EU), Spanish government and different industries. His major research areas are: parallel and distributed simulation, performance prediction and efficient management of multicluster–multicore systems and fault tolerance in parallel computers. He has supervised 19 Ph.D. thesis and co-authored more than 230 technical papers in journals and conference proceedings.