A Hierarchical Checkpointing Protocol for Parallel Applications in Cluster Federations

Sébastien Monnet, Christine Morin, Ramamurthy Badrinath

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

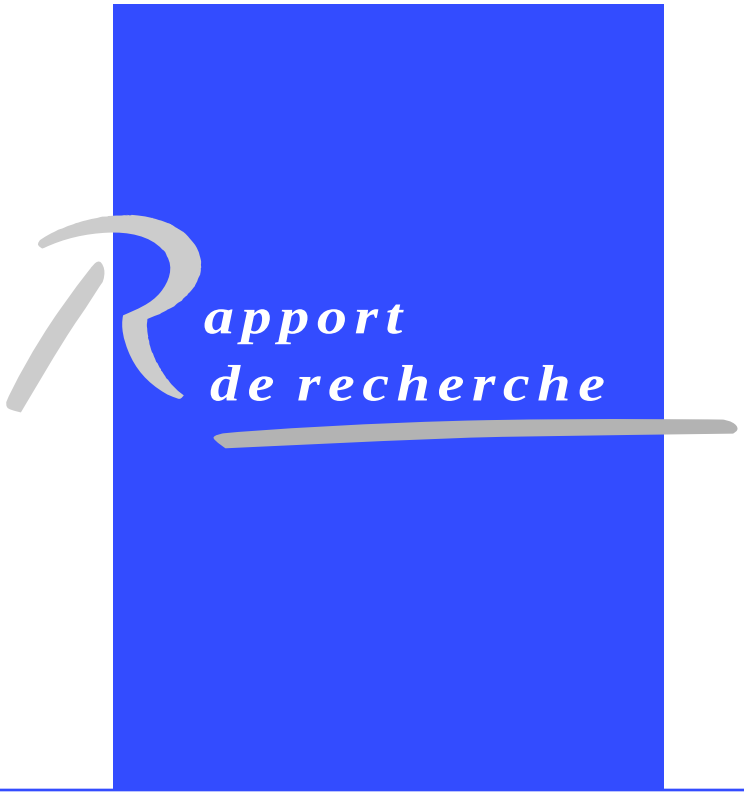
A Hierarchical Checkpointing Protocol for Parallel Applications in Cluster Federations

Sébastien Monnet and Christine Morin and Ramamurthy Badrinath

N°5091

Janvier 2004

THÈME 1



ISSN 0249-6399 ISRN INRIA/RR--5091--FR+ENG

A Hierarchical Checkpointing Protocol for Parallel Applications in Cluster Federations

Sébastien Monnet * and Christine Morin † and Ramamurthy Badrinath ‡

Thème 1 — Réseaux et systèmes
Projet PARIS

Rapport de recherche n ° 5091 — Janvier 2004 — 24 pages

Abstract:

Code coupling applications can be divided into communicating modules, that may be executed on different clusters in a cluster federation. As a cluster federation comprises of a large number of nodes, there is a high probability of a node failure. We propose a hierarchical checkpointing protocol that combines a synchronized checkpointing technique inside clusters and a communication-induced technique between clusters. This protocol fits to the characteristics of a cluster federation (large number of nodes, high latency and low bandwidth networking technologies between clusters). A preliminary performance evaluation performed using a discrete event simulator shows that the protocol is suitable for code coupling applications.

Key-words: Cluster Federation, Checkpointing and Recovery, Fault-tolerance, Parallel Application, Code Coupling

(Résumé : tsvp)

* Sebastien.Monnet@irisa.fr
† Christine.Morin@irisa.fr
‡ badrinar@india.hp.com

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Un protocole hiérarchique de sauvegarde de points de reprise d'applications parallèles dans les fédérations de grappes de calculateurs

Résumé : Les applications de type couplage de codes peuvent être divisées en modules communicants pouvant s'exécuter sur différentes grappes d'une fédération. Une fédération de grappes de calculateurs comportant un grand nombre de nœuds, la probabilité de défaillance d'un nœud est très élevée. Nous proposons un protocole de sauvegarde de points de reprise hiérarchique qui combine une technique de sauvegarde coordonnée de points de reprise au sein d'une grappe et une technique de sauvegarde de points de reprise induits par les communications entre les grappes. Ce protocole convient aux caractéristiques d'une fédération de grappes de calculateurs (grand nombre de nœuds, latence élevée et faible débit de communication entre les grappes). Les premières évaluations réalisées à l'aide d'un simulateur à événements discrets montrent que le protocole est adapté aux applications de type couplage de codes.

Mots-clé : fédération de grappes, points de reprise, tolérance aux fautes, applications parallèles, couplage de codes

1 Introduction

Cluster federations contain a large number of nodes and are heterogeneous. Nodes in a cluster are often linked by a SAN (System Area Network) while clusters are linked by LANs (Local Area Network) or WANs (World Area Network). The applications running on such architectures are often divided into communicating modules. These modules may need to run on different clusters for various reasons: security, hardware or software constraints, or because the application needs a very large number of nodes. An example of such applications is one coupling several simulation codes that sometimes need to communicate with each other.

The literature describes a lot of checkpoint/restart protocols suitable for clusters but very few work has been done to adapt these protocols to large scale architectures such as cluster federations and to take benefit of the communication patterns of code coupling applications.

We propose a hierarchical checkpointing protocol which has a limited impact on the performance of code coupling applications executed in a cluster federation. As SAN networks used in clusters exhibit low latency and high throughput, a coordinated checkpointing approach can be used to efficiently checkpoint the state of the processes executing inside a cluster. As networks used for interconnecting clusters in a cluster federation are LANs or WANs with a much higher latency and a lower bandwidth than SANs, a coordinated checkpointing approach cannot be used at the federation level. The hierarchical protocol we propose relies on a communication-induced checkpointing strategy to build a global checkpoint of a code coupling application executed on several clusters of a cluster federation. A communication-induced checkpointing approach is reasonable for code coupling applications as inter-module communications are not very intensive.

Our protocol, which is called HC^{3}I checkpointing protocol thereafter, has been simulated using a discrete event simulator. Preliminary results show that it works well with applications like code coupling.

The remainder of this paper is organized as follows. Section 2 present the protocol design principles. Section 3 describes the hierarchical protocol combining coordinated and communication-induced checkpointing techniques in a cluster federation. Section 4 presents some of the algorithms, Section 5 shows a sample execution with the HC^{3}I checkpointing protocol. Section 6 gives a brief description of the discrete event simulator we used for the evaluation of the HC^{3}I protocol and analyzes preliminary performance results. In Section 7, our work is compared with related works. Section 8 concludes.

2 Design Principles

2.1 Models and assumptions

Application model. We consider parallel applications designed using the code coupling model. Processes of this kind of applications are divided into groups (modules). Processes inside the same group communicate a lot while communications between processes belonging

to different groups are limited. Inter-group communications may be pipelined as in Figure 1 or they may consist of exchanges between two modules for example.

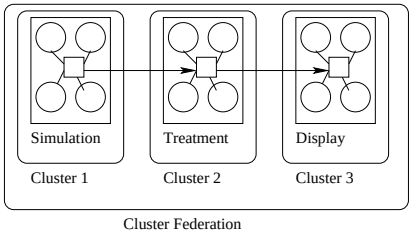


Figure 1: *Application Model*

System model. We assume the following system model. As shown in Figure 2, a *node* is a system-level module that implements the protocol. It is able to save the processes states, to catch every inter-processes message, and to communicate with other nodes for protocol needs.

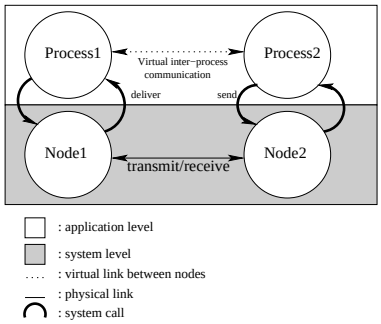


Figure 2: *System Model*

Architecture model and network assumptions. We assume a cluster federation as a set of clusters interconnected by a WAN, inter-cluster links being either dedicated or even Internet, or a LAN. Such an architecture is suitable for the code coupling application model described above. Each group of processes may run in a cluster where network links have small latencies and large bandwidths (SAN). We assume that a sent message will be received in an arbitrary but finite laps of time. This means that the network is reliable, it does not lose messages. This assumption implies that the fault tolerance mechanism has to take care of in-transit messages, since they are assumed not to be lost.

Failure assumptions. We assume that only one fault occurs at a time. However, the protocol can be extended to tolerate simultaneous faults as explain in Section 8. The failure model is fail-stop. When a node fails it will not send messages anymore. The protocol takes into account neither omission nor Byzantine faults.

2.2 Checkpointing large scale applications in cluster federations

Dependencies and consistent state. The basic principle of all backward error recovery techniques is to periodically store a consistent state of the application in order to be able to restart from there in the event of a failure. A process state consists of all the data it needs to be restarted (i.e. the virtual memory, list of opened files, sockets, etc.). A parallel application state is defined as the set of all its processes states. *Consistent* means that there is neither in-transit messages (sent but not received) nor ghost-messages (received but not sent) in the set of process states stored.
Messages generate dependencies. For example, Figure 3 presents the execution of two processes which both store their local states (*S1* and *S2* respectively). A message m is sent from process 1 to process 2. If the execution is restarted from the set of states *S1/S2*, the message m will have been received by process 2 but not sent by process 1 (ghost-message). Process 1 will send m again to process 2 which is not consistent. The State *S2* can depend on the content of m which may depend on the state *S1*.

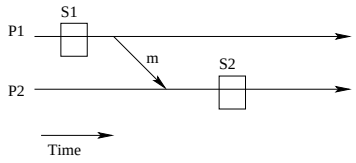


Figure 3: *Dependency between two states*

The most recent record of a consistent state is called the *recovery line.*

Checkpointing methods. The recovery line can be found at checkpoint time (i.e. when the states are stored). This method is called coordinated checkpointing. This means there is a two-phase commit protocol during which application messages are frozen. With the independent checkpointing method, each process of a parallel application can store its local state without any synchronization. The recovery line is computed at rollback time. Quasi-synchronous methods also exist. For example the application messages can be used to piggy-back some more information in order for the receiver to know if it needs to store its local state. This last method is called communication-induced checkpointing. [5] provides detailed information about these different checkpointing techniques.

Large scale checkpointing The large number of nodes and network performance between clusters do not allow a global synchronization. An independent checkpointing mechanism does not fit either: tracking dependencies to compute the recovery line at rollback time would be very hard and nodes may rollback to very old checkpoints (domino effect). If we intend to log inter-cluster communications (to avoid inter-cluster dependencies), we need the piecewise deterministic (PWD) assumption. The PWD assumption means that we are able to replay a parallel execution in a cluster that produces exactly the same messages as the first execution. This assumption is very strong. Replaying a parallel execution means detecting, logging and replaying all non-deterministic events, which is very difficult.

Hierarchical checkpointing Inside a cluster we use a coordinated checkpointing method. It ensures that the stored state (the cluster checkpoint) is consistent. Coordinated checkpointing is possible inside a cluster as nodes are interconnected with a high performance network (low latency and large bandwidth). Coordinated checkpointing is a well-established technique [8], [4], [11], [1] which is relatively easy to implement. A Cluster Level Checkpoint is called CLC thereafter. The assumption that the number of inter-cluster messages is low leads us to use a communication-induced method between clusters. This means each cluster takes CLC independently, but information is added to each inter-cluster communication. It may lead the receiver to take a CLC (called forced CLC) to ensure the recovery line progress. Therefore we propose a hierarchical protocol combining coordinated and communication-induced checkpointing (HC^3I).

3 Description of the HC^3I Checkpointing Protocol

This section presents the HC^3I checkpointing protocol, the algorithms can be found in [7].

3.1 Cluster level checkpointing

In each cluster, a traditional two-phase commit protocol is used. An initiator node broadcasts (in its cluster) a *CLC request* (algorithm 3). All the cluster nodes acknowledge the request, then the initiator node broadcasts a *commit*. Algorithm 4 represents our implementation of the two-phase commit protocol. Between the request and the commit messages, application messages are queued to prevent intra-cluster dependencies (algorithm 5). In order to be able to retrieve CLC data in case of a node failure, each node record its part of the *CLCs*, and in the memory of an other node in the cluster. Because of this stable storage implementation, only one simultaneous fault in a cluster is tolerated. Each CLC is numbered. Each node in a cluster maintains a sequence number (SN). SN is incremented each time a CLC is committed. This ensures that the sequence number is the same on all the nodes of a cluster (outside the two-phase commit protocol). The SN is used for inter-cluster dependency tracking. Indeed, each cluster takes its CLC periodically, independently from the others.

3.2 Federation level checkpointing

In our application model, communications between two processes in different clusters may appear. This generates dependencies between *CLCs* taken in different clusters. Dependencies need to be tracked in order to allow the application to be restarted from a consistent state. Forcing a *CLC* in the receiver's cluster for each inter-cluster application message would work but the overhead would be huge as it would force useless checkpoints. In Figure 4, cluster 2 takes two forced *CLCs* (the filled ones) at message reception, and the application takes messages into account only when the forced *CLC* is committed. *CLC2* is useful: in the event of a failure, a rollback to *CLC1/CLC2* will be consistent (*m1* would be sent and received again). On the other hand, forcing *CLC3* is useless: cluster 1 has not stored any *CLC* between its two message sending. In the event of a failure it will have to rollback to *CLC1* which will force cluster 2 to rollback to *CLC2*. *CLC3* would have been useful only if cluster 1 would have stored a *CLC* after sending *m1* and before sending *m2*.

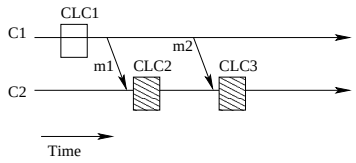


Figure 4: *Limitation of the number of forced CLCs*

Thus, a *CLC* is forced in the receiver's cluster only when a *CLC* has been stored in the sender's cluster *since the last communication from the sender's cluster to the receiver's cluster*. This is controlled using the *SN* (introduced in Section 3.1). The current cluster's sequence number is piggy-backed on each inter-cluster application message (Section 4.3 describes the message data structure). To be able to decide if a *CLC* needs to be initiated, all the processes in each cluster need to keep the last received sequence number from each other cluster. All these sequence numbers are stored in a *DDV* (Direct Dependencies Vector, [2]). Algorithm 6 shows how the receiver decides if it needs to initiate a forced *CLC*.
More formally: $DDV_j[i]$ is the i^{th} *DDV* entry of cluster j, and SN_i is the sequence number of cluster i.
For a cluster j:
If $i=j$, $DDV_j[i]=SN_j$
If $i \neq j$, $DDV_j[i]$= *last received* SN_i *(0 if none).*
Note that the size of the *DDV* is the number of clusters in the federation, not the number of nodes. In order to have the same *DDV* and *SN* on each node inside a cluster, we use the synchronization induced by the *CLC* two-phase commit protocol to synchronize them (as described by algorithm 4). Each time the *DDV* is updated, a forced *CLC* is initiated which ensures that all the nodes in the cluster which take a *CLC* will be timestamped by the same *DDV* at commit time.

3.3 Logs to avoid huge rollbacks

Coordinated checkpointing implies to rollback the entire cluster of a faulty node. We want to limit the number of clusters that rollback. If the sender of a message does not rollback while the receiver does, the sender's cluster does not need to be forced to rollback. When a message is sent outside a cluster, the sender logs it optimistically in its volatile memory (logged messages are used only if the sender does not rollback). This is shown by algorithm 5. The message is acknowledged with the receiver's *SN* which is logged along with the message itself (algorithm 7). The next section explains which messages are replayed in the event of a failure.

3.4 Rollback

When a node failure is detected, the cluster rolls back to its last stored *CLC* (the description of the failure detector is out of the scope of this paper). One node in each other cluster in the federation receives a *rollback alert*. It contains the faulty cluster's *SN* that corresponds to the *CLC* to which it rolls back. When a node receives such a *rollback* alert from another cluster with its new *SN*, it checks if its cluster needs to rollback by comparing its *DDV* entry corresponding to the faulty cluster to the received *SN*. If the former is greater than or equal to the latter its cluster needs to rollback to the first (the older) *CLC* which has its *DDV* entry corresponding to the faulty cluster greater than or equal to the received *SN*. The node that has received the alert initiates the rollback.
If a cluster needs to rollback due to a received alert, it sends a *rollback alert* containing its new *SN* to alert all the other clusters. This is how the recovery line is computed.
Even if its cluster does not need to rollback, a node receiving a *rollback alert* broadcasts it in its cluster. Logged messages sent to nodes in the faulty cluster acknowledged with a *SN* greater than the alert one (or not acknowledged at all) will then be resent.
Our communication-induced mechanism implies that clusters need to keep multiple *CLC* and logged messages. They need to be garbage-collected.

3.5 Garbage collection

Our protocol needs to store multiple *CLCs* in each cluster in order to compute the recovery line at rollback time. The memory cost may become important. Periodically, or when a node memory saturates, a garbage collection is initiated. Our garbage collector is centralized. A node initiates a garbage collection, it asks one node in each cluster to send back its list of all the *DDVs* associated with the stored *CLCs*. Then it simulates a failure in each cluster and keeps the smallest *SN* to which the clusters of the federation might rollback. It sends a vector containing all the smallest *SNs* to one node in each cluster which broadcasts it in its cluster.
Each node removes the *CLCs* which have their cluster *DDV* entry smaller than the smallest *SN* (received in the vector) associated to their cluster.
They also remove logged messages that are acknowledged with a *SN* smaller than the re-

ceiver's cluster smallest *SN*. Algorithm 8 about garbage collection messages is at the end of next section.

4 Algorithms

This section presents the main algorithms of the HC^{3}I Checkpointing Protocol. More details can be found in [6] (in French). To make it simple we introduce the notion of *leader*. In each cluster one primary *leader* and one secondary *leader* are chosen (in a static way at the initialization). These nodes are responsible for failure detection, restarting faulty nodes and inter-cluster protocol communications (rollback alert and garbage collection messages). The algorithms are not detailed, for example, *takeTentativeCkPt()* means storing the local state locally and on another node (and waiting for its acknowledgement).
Each cluster has a unique ID, and in each cluster, each node has also a unique rank.

4.1 Data structures

These are some data structures used in the algorithms.

Constants

- $nbClusters$ number of clusters.
- $myClusterId_i$ the ID of cluster i.
- $nbNodes_i$ the number of nodes in cluster i.
- $myRank_{i,j}$ the ID of node j in the cluster i.
- $lSet_i$ set of cluster i leaders.
- $otherLeaders_i$ set of the other clusters leaders - in each cluster, the leaders have to be able to communicate with the others.

Timers

- *iMALIVETimer* delay between heartbeats for the failure detection.
- *chCkAliveTimer* delay during which we should have receive at least one heartbeat from every node in a cluster.
- *tentativeCkPtTimer* maximum time between a *checkpoint request* and its corresponding *commit*.
- *waitForAllTimer* maximum time to wait after a *checkpoint request* for receiving all acknowledgments.

- *gCTimer* time between garbage collections.
- *ckPtTimer* time between two unforced *CLCs*.

Others

- $mySn_{i,j}$ the sequence number.
- $myDDV_{i,j}$ the *DDV*.
- $duringCkPt_{i,j}$ a boolean to know if a node is currently in the two phase commit protocol (i.e. checkpointing).
- $hb_{i,j}$ a vector with $nbNodes_i$ entries to remember the received heartbeats.
- $oldHb_{i,j}$ a copy of $hb_{i,j}$.
- $ckPtAckSet_{i,j}$ set of nodes that have acknowledge a *checkpoint request*.
- $gcAckSet_{i,j}$ set of nodes that have acknowledge a *garbage request*.
- $initiator_{i,j}$ rank of the last *CLC* initiator.

Logs Each node logs in volatile memory messages related to inter-cluster communications: the message itself, the receiver's ID, and the sequence number of the receiver (known by the message acknowledgement).

4.2 Initialization

Algorithm 1 is the initialization sequence, it is done by each node in the cluster federation at launch time. It sets the *DDV*, the sequence number, some variables and initialize some timers.

Algorithm 1: initialization

```
for i ← 0 to nbClusters do
    myDDV[i] ← 0;
mySn ← 0;
duringCkPt ← false;
launchTimer(iMALIVETimer );
launchTimer(ckPtTimer );
if ROLE = leader then
    launchTimer(chCkAliveTimer );
```

4.3 Messages structure

Messages exchanged by nodes have the following structure :

- *sender* the identity of the sender (its rank and cluster IDs)
  - *sender.rank*
  - *sender.clusterId*
- *type* (see *Message dispatching* algorithm).
- *subtype* (see *ckPtHandler* algorithm).
- *sn* the sender's sequence number.
- *data* the message itself.

In Section 3.2 ("Federation Level Checkpointing"), it is explained why messages need to contain *sn*, for dependencies tracking. It is used by the receiver to know if it needs to take a forced *CLC*. Algorithm 2 dispatches a message according to its type and its sender. Some of these algorithms are presented in the following of this section.

Algorithm 2: Messages Dispatching

```
Data        : m, the received message
if m.sender.siteId=myClusterId then
    message from the cluster;
    switch m.type do
        case CKPT
            ckPtHandler(m);
        case ROLLBACK
            rollbackHandler(m);
        case FD
            fdHandler(m);
        case GC
            gcHandler(m);
        case INTERNALALERT
            internalAlertHandler(m);
        otherwise
            just a normal application message from the same cluster - nothing to do;
            if duringCkPt =true then
                storeMessage(m);
            else
                deliver(m);
else
    message from another cluster in the federation;
    if duringCkPt =true then
        let the coordinated checkpoint finish;
        storeMessFromOut(m);
    else
        switch m.type do
            case ROLLBACKALERT
                the node is part of its clusters lSet;
                rollbackAlertHandler(m);
            case ACK
                ackFromOutsideHandler(m);
            otherwise
                just a normal application message;
                messFromOutsideHandler(m);
```

4.4 Checkpointing algorithms

This algorithm (3) initiates a *CLC* in a cluster, like it is explained in Section 3.1 ("Cluster Level Checkpointing").

Algorithm 3: initiateCkPt

```
initialization of some data structures;
ckPtAckSet ← ∅;
duringCkPt ← true;
initiator ← myRank;
take the node's own tentative checkpoint;
takeTentativeCkPt();
ask the other nodes in the cluster to do the same;
broadCastReqCkPt();
in case of failure during the checkpoint;
launchTimer(waitForAllTimer );
```

Algorithm 4 is executed when receiving checkpointing Messages. It is our implementation of the two phase commit protocol described in Section 3.1.
The names of the functions are used to describe what they do. For example, *launchTimer* launches a timer, and *sendCkPtAck(id,sn)* sends an acknowledgement (i.e. the type of the message will be "CKPT" and its subtype will be "ACK") with *sn* to the node represented by *id*.
We can see that the synchronization induced by the two phase commit protocol also synchronizes the DDV on all the cluster nodes.

Algorithm 4: ckPtHandler

```
Data        : m received from a node in the cluster
switch m.subType do
    case REQ
        the node is requested to take a tentative checkpoint;
        if duringCkPt =false then
            Not currently checkpointing;
            stopTimer(ckPtTimer )don't initiate a new checkpoint;
            initiator ← m.sender.rankremember the initiator's rank;
            duringCkPt ← true;
            takeTentativeCkPt();
            sendCkPtAck(m.sender,myDDV )acknowledgement;
            launchTimer(tentativeCkPtTimer );
        else
            the node is already in a checkpoint phase;
            if m.sender.rank≺initiator then
                only the one with the smallest rank is taken into account;
                if initiator =myRank then
                    the node was the other initiator;
                    stopTimer(waitForAllTimer );
                    removeCkPtAckSet();
                else
                    the node was not the other initiator;
                    stopTimer(tentativeCkPtTimer );
                initiator ← m.sender.rank;
                sendCkPtAck(m.sender,myDDV );
                launchTimer(tentativeCkPtTimer );
    case ACK
        If we receive an Ack, we are the initiator;
        add(ckPtAckSet,m.sender,receivedDDV );
        if ckPtAckSet =ALLINGRP then
            stopTimer(waitForAllTimer );
            computeNewDDV generate a new DDV in which entries are the max of the corresponding
            entries in all the DDVs;
            computeNewDDV(ckPtAckSet,newDDV );
            makeTentativePermanent();
            duringCkPt ← false;
            broadCastCkPtCommit(newDDV );
    case COMMIT
        stopTimer(tentativeCkPtTimer );
        makeTentativePermanent()this also increment mySn;
        myDDV ← newDDV;
        duringCkPt ← false;
        deliverAll()deliver all the waiting messages;
        replayMessFromOut();
        sendAll()send all the waiting messages;
        launchTimer(ckPtTimer );
```

4.5 Application messages transmission

Algorithm 5 is executed when a process is sending a message to another one in the cluster federation. The message is caught by the fault-tolerant layer. It puts the right message type and subtype, the sender identity and current sequence number... It checks if the message needs to be queued (if communications need to be frozen due to a current checkpoint being stored) or logged (if it's an inter-cluster message).

Algorithm 5: send

```
Data        : mess, the sent message
the message type and subtype are set by the function that call send, for example sendCkPtAck will set
type to CKPT and subtype to ACK;
if duringCkPt =true then
    froze communications during checkpointing;
    storeToSend(mess );
else
    mess.sn ← mySn;
    mess.sender.rank ← myRank;
    mess.sender.clusterId ← myClusterId;
    send the message on the network;
    transmit(mess );
    if it's an inter-cluster communication, log it;
    if receiver.clusterId ≠ myClusterId then
        the logging is optimistic and doesn't need stable storage;
        logVolatile(mess );
        its sn is suppose to be infinite (i.e. will have to be replayed) until the acknowledgment;
        logVolatile(∞);
```

Algorithm 6 is called when a message is coming from an other cluster. It checks if a *CLC* has to be initiated by comparing the received sequence number and its corresponding *DDV* entry as explained in Section 3.2. It acknowledges the messages with the appropriate sequence number.

Algorithm 6: messFromOutsideHandler

```
It comes from another cluster, check the dependences;
if m.sn≻myDDV [m.sender.clusterId] then
    acknowledge the message with the next sequence number: a checkpoint will be taken;
    acknowledgeMess(mySn ++);
    update the DDV;
    myDDV[m.sender.clusterId] ← m.sn;
    the message will be delivered at the commit;
    storeMessage(m );
    initiateCkPt();
else
    acknowledge the message with the current sequence number;
    acknowledgeMess(mySn );
    deliver(m );
```

Algorithm 7 represents the fact that inter-cluster are logged (by the sender) with the sequence number that the receiver has at reception time, as soon as they are acknowledged.

Algorithm 7: ackFromOutsideHandler

```
an inter-cluster send is acknowledged with the receiver current sn in the message data;
logVolatile(m.sn );
the received sn will replace the ∞ stored during the message logging phase;
```

4.6 Garbage collection

Algorithm 8 draws what is done for garbage collection. As described in section 3.5, initiating a garbage collection means sending a request for garbage collection to all the leaders in the federation. Then, this algorithm shows what does a node do when receiving such a request (it sends its entire *DDV* list, one *DDV* per *CLC* stored). When all the *DDV* lists have been received by the initiator, it computes the recovery line (explained in Section 3.5) then send it to all other leaders in the cluster federation. If a leader receives such a message it broadcasts it in its cluster and every node collects all its obsolete data.

Algorithm 8: gcHandler

```
switch m.subtype do
    case REQ
        sendGCACK(myDDV );
    case ACK
        add(gcAckSet,m.sender,receivedDDV );
        if gcAckSet.size=nbClusters then
            the initiator has received all the DDVs;
            stopTimer(gCTimer );
            computeRecoveryLine(gcAckSet,recoveryLine );
            sendCollect(otherLeaders,recoveryLine );
    case COLLECT
        if ROLE = _LEADER then
            broadcastCollect(recoveryLine );
        clean(recoveryLine );
```

5 Example

Figure 5 shows a sample execution on three clusters. It is composed of three successive snapshots of the execution. On each snapshot, the execution time goes from left to right, each horizontal line represents a parallel execution on a cluster. The boxes stand for the *CLCs*, the darker ones are forced *CLCs*. The corresponding *DDVs* are embedded in the *CLC*'s boxes.

The first snapshot shows a normal execution until a failure appears in cluster 2. Notice that each cluster stores a first *CLC* which is the beginning of the application. Cluster 1 sends message *m1* to cluster 2, it sends its *SN* (1) along with *m1*. When receiving *m1*, cluster 2 compares the received *SN* with cluster 1 *DDV* entry (0). 1 is greater than 0, this forces cluster 2 to take a *CLC* before delivering *m1* to the application level. When receiving *m2* from cluster 1, cluster 2 does not have to initiate a *CLC*, the received *SN* (1) is equal to cluster 1 *DDV* entry in cluster 2. As for *m1*, we see that *m3*, *m4* and *m5* force *CLCs* respectively on clusters 3, 3 and 1. Notice that inter cluster messages are acknowledged with the local *SN + 1* (the inter-cluster message will be delivered after the *CLC* is committed). Logged messages are not represented to keep the figure easy to read.

When a fault is detected in cluster 2, the whole cluster rolls back to its last stored *CLC*, its new *SN* is 3. It then sends a *rollback alert* with the *SN* 3 (second snapshot). Cluster

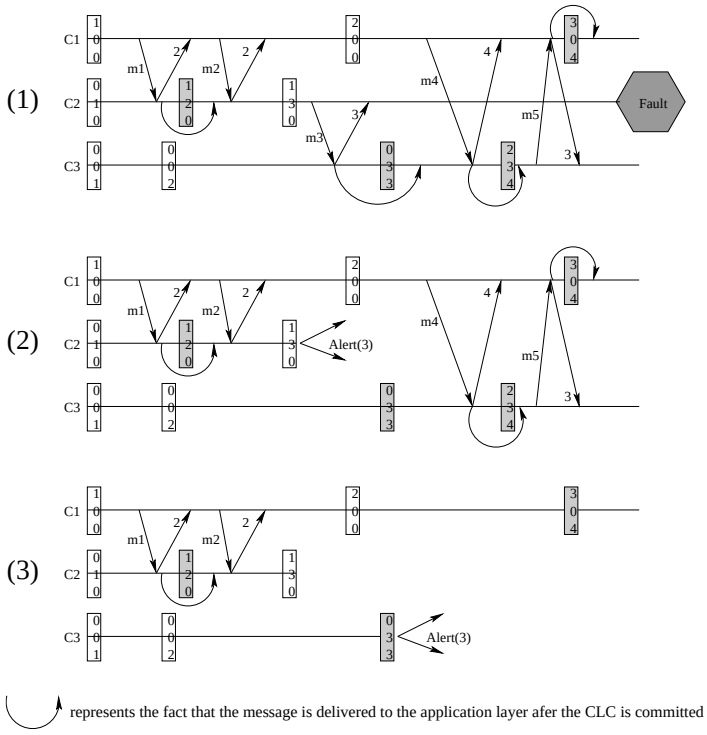


Figure 5: HC^{3}I checkpointing protocol sample

1 does not have any cluster 2 *DDV* entry greater than or equal to the received *SN* in its *DDVs* stored with the *CLCs*, it does not need to rollback. On the over hand, cluster 3 has to rollback to the first *CLC* that has its associated *DDV* containing cluster 2 entry greater than or equal (equal in the sample) to the received *SN*. Cluster 3 sends an alert with its new *SN*, 3 (third snapshot).
Cluster 2 has never received messages from cluster 3 so its *DDVs* entries corresponding to cluster 3 are all equal to 0. It does not need to rollback. Cluster 1 has to rollback to its last *CLC* which has 4 in cluster 3's entry. It sends a rollback alert with its new *SN* (3) but no cluster has to rollback anymore (due to the *DDV* lists).

6 Evaluation

To evaluate the protocol, a discrete event simulator has been implemented. We evaluate the protocol overhead in terms of network and storage cost first, then we observe what happens

with different communication patterns. Finally the garbage collector effectiveness and cost are evaluated.

6.1 Simulator

We use the C++SIM library [12] to write the simulator. This library provides generic threads, a scheduler, random flows and classes for statistical analysis. Our simulator is configurable. The user has to provide three files: a *topology file*, an *application file* and a *timer file*. The *topology file* specifies the number of clusters, the number of nodes in each cluster, the bandwidth and latency in each cluster and between clusters (represented as a triangular matrix) and the federation *MTBF* (Mean Time Between Failures). The *application file* contains, for each cluster, the mean computation time for each node, communication patterns between computations (represented by probabilities between nodes) and the application total time. Finally, the *timers file* contains the delays for the protocol timers for each cluster (delays between two *CLCs*, garbage collection, ...).
The simulator is composed of four main threads. The thread *Nodes* takes the identity of all the nodes, one by one. The thread *Network* stores the messages and computes their arrival time. The thread *Timers* simulates all the different timers. The thread *Controller* controls the other threads (launches them, displays results at the end,...). Communication between threads is done by shared variables.
The simulator can be compiled with different trace levels. With the higher trace level, we can observe each node time-stamped action (sends, receives, timer interruptions, log searches...). The lowest simulator output is statistical data, as messages count in clusters and between each cluster, number of stored *CLCs*, number of protocol messages,...

6.2 Network traffic and storage cost induced by the checkpointing protocol

Evaluating network traffic and storage cost is very hard. It depends on how the protocol is tuned. If the frequency of unforced *CLCs* is low in a cluster, the *SNs* will not grow too fast, so inter-cluster messages from this cluster would have a low probability to force *CLCs*. Reducing the protocol overhead becomes easy. If no *CLC* is initiated, the only protocol cost consists in logging optimistically in volatile memory inter-cluster messages and transmitting an integer (*SN*) with them. There is also a little overhead due to message interception (between the network interface and the application).
To take advantage of the protocol, the timer that regulates the frequency of unforced *CLCs* in a cluster should be set to a value that is much smaller than the MTBF of this cluster.
The first simulation evaluates how much *CLCs* the protocol forces. The simulator simulates 2 clusters of 100 nodes. In both clusters the network is "Myrinet-like" (10μs latency and 80Mb/sec bandwidth). The clusters are linked by "Ethernet-like" links (150μs latency and 100Mb/sec bandwidth). The application total execution time is 10 hours. There are lots of communications inside each cluster and few between them. This could correspond to a simulation running on cluster 0 and to trace processor on cluster 1, for example. Table 6.2

Sender's Cluster	Receiver's Cluster	Message Count
Cluster 0	Cluster 0	2920
Cluster 1	Cluster 1	2497
Cluster 0	Cluster 1	145
Cluster 1	Cluster 0	11

Table 1: Application messages

displays the number of messages (intra and inter-cluster).

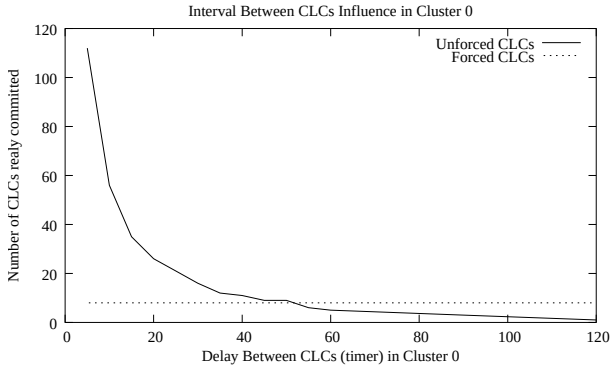


Figure 6: *Number of CLCs in Cluster 0*

Graph 6 and 7 show the number of forced and unforced committed *CLCs* in each cluster according to the delay between unforced *CLCs* in cluster 0 (x axis, in minutes). Cluster 1 delay between *CLCs* is set to infinite. Cluster 0 stores some forced *CLCs* (8) because of the communications from cluster 1. This number of forced *CLCs* is constant - there are few messages from cluster 1. Notice that the total number of stored *CLCs* is smaller than $\frac{totalcomputationtime}{delaybetweenCLCs}$ + *number of forced CLCs* because the timer is reset when a forced *CLC* is established. Clusters store a few more *CLCs*, but they are placed better (in time). Cluster 1 does not store any unforced *CLCs* as its timer is set to infinite, but it stores some forced *CLCs* induced by incoming communications from cluster 0. The number of these forced *CLCs* is proportional to the number of *CLCs* stored in cluster 0, because numerous messages come from cluster 0.
One may want to store more *CLCs* in cluster 1, if this cluster is intensively used and computation time is expensive for example. Graph 8 shows that cluster 0 (which "delay between *CLCs*" timer is set to 30 minutes) do not store more *CLCs* even if cluster 1 timer

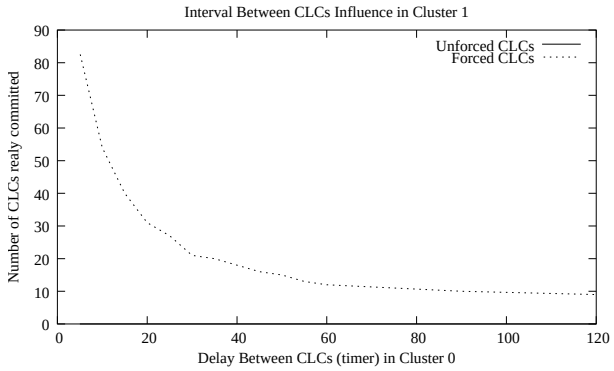


Figure 7: *Number of CLCs in Cluster 1*

is set to 15 minutes. This is thanks to the low number of messages from cluster 1 to cluster 0.

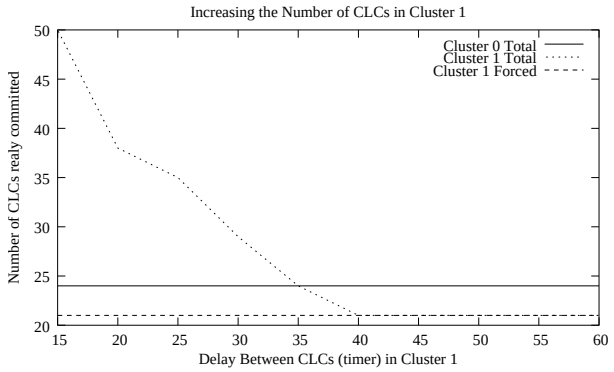


Figure 8: *Impact of the Number of CLCs*

6.3 Communication patterns

To better understand the influence of the communications patterns on the checkpointing protocol, Graph 9 shows what happens when the number of messages from cluster 1 to cluster 0 increases. Both cluster "delay between *CLCs*" timers are set to 30 minutes. The application is the same as in previous section except for the number of messages from cluster 1 to cluster 0 that is represented on the x axis.

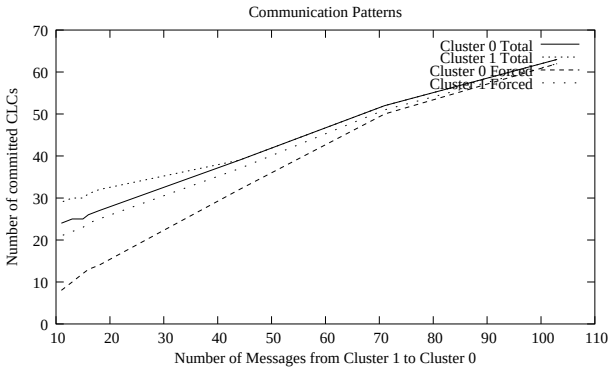


Figure 9: *Increasing Communication from Cluster 1 to Cluster 0*

The number of forced *CLCs* increases fast with the number of messages from cluster 1 to cluster 0. If the two clusters communicate a lot in both ways, *SNs* will grow very fast and most of the messages will induce a forced *CLC*. The overhead of our protocol will not be good in that case.

6.4 Garbage collection

A garbage collection has got a non negligible overhead. If N is the number of clusters in the federation, each garbage collection implies: N-1 inter-cluster requests; N-1 inter-cluster responses which contain the list of all the *DDVs* associated to the stored *CLCs* in a cluster; N-1 inter cluster collect requests; broadcast in each cluster.

However, our hybrid checkpointing protocol may store multiple *CLCs* in each cluster. They can become very numerous. It also logs every inter-cluster application message. We evaluate the efficiency of the garbage collector. For the sample above, in the case of 103 messages sent from cluster 1 to cluster 0, without any garbage collection, 63 *CLCs* are stored in each cluster. It means that each node in the federation stores 126 local states (its own 63 local states and the ones of one of its neighbor, because of the stable storage implementation).
If a garbage collection is launched every 2 hours, the maximum number of stored *CLCs* just after a garbage collection is 2 per cluster in this sample. Only oldest *CLCs* are removed, as explain in Section 3.5, rollbacks will not be too deep. The maximum number of logged messages during the execution in the sample above is 4 in both clusters. Table 2 shows for each garbage collection the number of *CLCs* stored just before and just after the collection.

In order to see what happens with more clusters, a second experimentation simulates an application that runs on three clusters. Clusters 0 and 1 have the same configuration as above. Cluster 2 is a clone of cluster 1. There's approximately 200 messages that leave and

Cluster 0 Before	Cluster 0 After	Cluster 1 Before	Cluster 1 After
10	2	11	2
18	2	18	2
15	2	14	2
14	2	15	2

Table 2: Number of stored *CLCs*

Cluster 0 (before)	30	48	54	38
Cluster 0 (after)	2	2	2	2
Cluster 1 (before)	50	80	78	64
Cluster 1 (after)	2	2	2	2
Cluster 2 (before)	50	80	78	64
Cluster 2 (after)	2	2	2	2

Table 3: Number of stored *CLCs*

arrive in each cluster. Table 3 shows for each garbage collection the number of *CLCs* stored just before and just after the collection. After each garbage collection only 2 *CLCs* are kept. Thanks to the communication-induced method, the recovery line progresses. A tradeoff has to be found between the frequency of garbage collection and the number of *CLCs* stored.

7 Related Work

A lot of papers about checkpointing methods can be found in the literature. However, most of the previous works are related to clusters, or small scale architectures. A lot of systems are implemented at the application level, partitioning the application processes into steps. Our protocol is implemented at system level so that programmers do not need to write specific code. Moreover the protocol in this paper takes clusters federation architecture into account. This section presents several works that are close to ours.

Integrating fault-tolerance techniques in grid applications. [9] does not present a protocol for fault tolerance but it describes a framework that provides hooks to help developers to incorporate fault tolerance algorithms. They have implemented different known fault tolerance algorithms and it seems to fit well with large scale. However, these algorithms are implemented at application level and are made for object-based grid applications.

MPICH-V. [3] describes a fault tolerant implementation of MPI. It is designed for large scale architectures. All the communications are logged and can be replayed. This avoids

all dependencies so that a faulty node will rollback, but not the others. But this means that strong assumptions upon determinism have to be made. Our protocol does not need any assumption upon the application determinism, moreover it takes advantage of the fast network available in the clusters.

Hierarchical coordinated checkpointing. The work presented in [10] is the closest to ours. It proposes a coordinated checkpointing method, based on the two-phase commit protocol. The synchronization between two clusters (linked by slower links) is relaxed. In [10], it is the coordinated checkpointing mechanism that is relaxed between clusters. It is not a hybrid protocol like ours. Our protocol is more relaxed, it is "independent checkpointing" if there are no inter-cluster messages.

8 Conclusion and Future Work

This paper introduces a hierarchical checkpointing protocol suitable for code coupling applications. It relies on a hybrid method combining coordinated checkpointing inside clusters and communication-induced checkpointing between clusters. The protocol can be tuned according to the underlying network, the application communication patterns and needs. The dependency tracking mechanism can be improved by adding some transitivity (by sending the whole DDV instead of the SN) in order to take less forced checkpoints. The user should be able to choose the degree of replication in the stable storage implementation inside a cluster (in order to tolerate more than one fault in a cluster). The protocol should tolerate simultaneous faults in different clusters (the garbage collector should take care of this). At last, the garbage collector could be more distributed. We need to implement the protocol on a real system to validate it.

Acknowledgements

The authors wish to thank Gabriel Antoniu for his careful proof-reading of this paper.

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. *The Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 167–176, August 1999.

[2] R. Badrinath and C. Morin. Common mechanisms for supporting fault tolerance in DSM and message passing systems. Technical report, July 2003.

[3] G. Bosilca, A. Bouteiller, F. Cappello, S. Djailali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Proceedings of the IEEE/ACM SC2002 Conference*, pages 29–47, Baltimore, Maryland, November 2002.

[4] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. In *Operating Systems Design and Implementation*, pages 59–73, October 1996.

[5] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys (CSUR)*, 34:375–408, September 2002.

[6] S. Monnet. Conception et évaluation d'un protocole de reprise d'applications parallèles dans une fédération de grappes de calculateurs. Rapport de stage de DEA, IFSIC, Université de Rennes 1, France, June 2003. In French.

[7] S. Monnet, C. Morin, and R. Badrinath. A hierarchical checkpointing protocol for parallel applications in cluster federations. Research report 1592, IRISA, Rennes, France, Jan. 2004.

[8] C. Morin, A.-M. Kermarrec, M. Banâtre, and A. Gefflaut. An Efficient and Scalable Approach for Implementing Fault Tolerant DSM Architectures. *IEEE Transactions on Computers*, 49(5):414–430, May 2000.

[9] A. Nguyen-Tuong. *Integrating Fault-Tolerance Techniques in Grid Applications*. PhD thesis, Faculty of the School of Engineering and Applied Science at the University of Virginia, August 2000.

[10] H. Paul, A. Gupta, and R. Badrinath. Hierarchical Coordinated Checkpointing Protocol. In *International Conference on Parallel and Distributed Computing Systems*, pages 240–245, Novenber 2002.

[11] J. Rough and A. Goscinski. Exploiting Operating System Services to Efficiently Checkpoint Parallel Applications in GENESIS. *Proceedings of the 5th IEEE International Conference on Algorithms and Architectures for Parallel Processing*, October 2002.

[12] C++SIM. `http://cxxsim.ncl.ac.uk`.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399