HydEE: Failure Containment without Event Logging for Large Scale Send-Deterministic MPI Applications

Amina Guermouche, Thomas Ropars, Marc Snir, Franck Cappello

HydEE: Failure Containment without Event Logging for Large Scale Send-Deterministic MPI Applications

Amina Guermouche*, Thomas Ropars†, Marc Snir‡, Franck Cappello*‡
*INRIA Saclay-Île de France, F-91405 Orsay, France
†École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
‡University of Illinois at Urbana-Champaign, Urbana, IL, USA
Email: guermou@lri.fr, thomas.ropars@epfl.ch, snir@illinois.edu, fci@lri.fr

***Abstract*—High performance computing will probably reach exascale in this decade. At this scale, mean time between failures is expected to be a few hours. Existing fault tolerant protocols for message passing applications will not be efficient anymore since they either require a global restart after a failure (checkpointing protocols) or result in huge memory occupation (message logging). Hybrid fault tolerant protocols overcome these limits by dividing applications processes into clusters and applying a different protocol within and between clusters. Combining coordinated checkpointing inside the clusters and message logging for the inter-cluster messages allows confining the consequences of a failure to a single cluster, while logging only a subset of the messages. However, in existing hybrid protocols, event logging is required for all application messages to ensure a correct execution after a failure. This can significantly impair failure free performance. In this paper, we propose HydEE, a hybrid rollback-recovery protocol for send-deterministic message passing applications, that provides failure containment without logging any event, and only a subset of the application messages. We prove that HydEE can handle multiple concurrent failures by relying on the send-deterministic execution model. Experimental evaluations of our implementation of HydEE in the MPICH2 library show that it introduces almost no overhead on failure free execution.**

***Keywords*-High performance computing, MPI, fault tolerance, send-determinism, failure containment**

I. Introduction

All studies about future Exascale systems consider that fault tolerance is a major problem [20]. At such a scale, the mean time between failures is expected to be between few hours and 1 day. The International Exascale Software Project (IESP) roadmap mentions extending the applicability of fault tolerance techniques towards more local recovery as one of the main research directions [12].

Fault tolerance for message passing applications, including MPI (Message Passing Interface [23]) applications, is usually provided through rollback-recovery techniques [14]. However, existing rollback-recovery protocols have severe scalability limitations. Coordinated checkpointing protocols force the rollback of all processes to the last coordinated checkpoint in the event of a failure. Message logging protocols require storing all application message payloads. One could think of replication as an alternative to rollback-recovery [27]. But replicating the workload is very expensive with respect to resources and energy consumption.

Failure containment, *i.e.*, limiting the consequences of a failure to a subset of the processes, is one of the most desirable properties for a rollback-recovery protocol targeting very large scale executions [13]: i) it can reduce energy consumption by limiting the amount of rolled back computation; ii) it can speed up recovery because recovering a subset of the processes is faster than recovering the whole application [26]; iii) it can improve the overall system utilization because the computing resources that are not involved in the recovery could be used by other applications meanwhile.

Failure containment in message passing applications is provided by logging messages to avoid rollback propagation. Pessimistic or causal message logging protocols provide perfect failure containment, since they only require the failed processes to roll back after a failure. However it comes at the expense of logging all application messages, usually in the nodes memory [19]. Hybrid rollback-recovery protocols can be used to provide failure containment without logging all messages.

Hybrid rollback-recovery protocols have been proposed as a way to combine the advantages of two rollback-recovery protocols [31]. They are based on application processes clustering to apply one protocol inside each cluster (local level) and a different protocol between clusters (global level). Using coordinated checkpointing at the local level and message logging at the global level allows to limit the consequences of a failure to a single cluster while logging only inter-cluster messages [32], [22], [8]. Such an approach fits well the communication pattern of most High Performance Computing (HPC) applications. It has been shown, on a large variety of MPI applications, that a single failure can be confined to less than 15% of the processes by logging less than 15% of the messages [28].

In the usual piecewise deterministic execution model, Bouteiller *et al.* showed that it is mandatory to log all non deterministic events reliably during failure free execution, to be able to correctly recover an application from a failure using a hybrid coordinated checkpointing/message logging

protocol [8]. However, event logging can impair failure free performance, even when implemented in a distributed way [29].

The send deterministic execution model is a new execution model that holds for most HPC MPI applications [10]. It states that, considering a given set of input parameters for an application, the sequence of messages sent by each process is the same in any correct execution of the application. This new model allows to design new rollback-recovery protocols [16].

To deal with the applicability of rollback-recovery techniques to message passing applications execution at extreme scale, this paper presents the following contributions:

- We propose HydEE, a hybrid rollback-recovery protocol for send-deterministic applications that combines coordinated checkpointing and message logging. We provide a detailed description of HydEE, including pseudo-code.
- We show that HydEE can tolerate multiple concurrent failures without logging any non deterministic event during failure free execution. To our knowledge, it is the first rollback-recovery protocol to provide failure containment for non fully deterministic applications without relying on a stable storage[1].
- We present an evaluation of our implementation of HydEE in the MPICH2 library. Experiments run on a set of HPC benchmarks over a Myrinet/MX high performance network show that HydEE provides at most 2% performance overhead on failure free execution.

HydEE is a good candidate for fault tolerance at exascale because it requires to store only a subset of the application messages content in the computing nodes local storage (not persistently) to provide failure containment.

The paper is organized as follows. Section II describes the execution model considered in this paper. It outlines the impact of send-determinism on rollback-recovery protocols design. We provide a detailed description of HydEE in Section III, and prove that it can tolerate multiple concurrent failures in Section IV. Section V presents our experimental results. Then, we compare HydEE to the related work in Section VI. Finally, we draw the conclusions and present some future works in Section VII.

II. Modeling the Determinism of a Message Passing Execution

The design of a rollback-recovery protocol is strongly impacted by the execution model that is assumed. In this section, we introduce the main classes of rollback-recovery protocols, *i.e.* checkpointing protocols and message logging protocols, by studying the execution model they consider. We outline the impact of the send-deterministic model and explain how it applies to MPI applications.

[1]Processes checkpoints are saved on stable storage but are not required for failure containment

A. Message Passing System Model

To model a message passing parallel execution, we consider a set $P = \{p_1, p_2, ..., p_n\}$ of n processes, and a set C of channels connecting any ordered pair of processes. Channels are assumed to be FIFO and reliable but no assumption is made on system synchrony.

An execution E is defined by an initial state $\Sigma^0 = \{\sigma_1^0, \sigma_2^0, ..., \sigma_n^0\}$, where σ_i^0 is the initial state of process p_i, and a sequence $S = e_1, e_2, e_3...$ of events. An event changes the state of a process. Event e_i^k is the k^{th} event on process p_i. The state of process p_i after the occurrence of e_i^k is σ_i^k.

An event can be the sending of a message ($send(m)$), the reception of a message ($recv(m)$), or a local event. The events in S are partially ordered by the Lamport's *happened-before* relation [21], denoted $\rightarrow$.

The sub-sequence of S consisting of events on process p_i is denoted $S|p_i$. The state σ_i^k of process p_i can be defined as $\left(\sigma_i^0, S|p_i^k\right)$, where $S|p_i^k = e_i^1, e_i^2, ..., e_i^k$. Checkpointing process p_i consists in saving a state σ_i on a reliable storage. A global state Σ is composed of one state of each process in P, $\Sigma = \{\sigma_1^{k_1}, \sigma_2^{k_2}, ..., \sigma_n^{k_n}\}$. Event $e_i^k \in \Sigma$ if $\sigma_i^l \in \Sigma \; and \; k \leq l$. A process state σ_i^k is final if no transition is possible to σ_i^{k+1}. A global state $\Sigma = \{\sigma_1^{k_1}, \sigma_2^{k_2}, ..., \sigma_n^{k_n}\}$ is final if all $\sigma_j^{k_j}$ are final.

In this paper, we consider a fail-stop failure model for the processes and assume that multiple concurrent failures can occur. A rollback-recovery protocol ensures that the execution of a message passing application is correct despite failures. Execution $E = \left(\Sigma^0, S\right)$ is correct if and only if:

- the sequence of events in S is consistent with the *happened-before* relation;
- the global state at the end of the execution is final.

After a failure, a rollback-recovery protocol tries to recover the application in a consistent global state. The global state Σ is consistent iff for all events e, e':

$$e' \in \Sigma \; and \; e \rightarrow e' \Longrightarrow e \in \Sigma \tag{1}$$

A message is said *orphan* in the global state Σ if $recv(m) \in \Sigma$ but $send(m) \notin \Sigma$. A consistent global state is a state without orphan messages.

B. Modeling Applications Determinism

Rollback-recovery protocols are defined in a model that includes a specification of the application determinism. To reason about the determinism of an application, we need to consider $\mathscr{E}$, the set of correct executions from an initial application state Σ^0. The set $\mathscr{S}$ includes the sequences of events S corresponding to the executions in $\mathscr{E}$.

Checkpointing protocols are based on process checkpointing. They do not make any assumption on the determinism of the applications. They consider a non deterministic execution model.

***Definition** 1 (Non deterministic execution model):* An application execution is not deterministic if, considering an initial state Σ^0, $\exists S \; and \; S' \in \mathscr{S}$ and a process $p \in P$ such that:

$$S|p \neq S'|p \tag{2}$$

It implies that, after a failure, the application has to be restarted from a consistent global state: Any process state that depends on an event that is not included in the last checkpoint of one failed process has to be rolled back. That is why no checkpointing protocol can provide failure containment.

Checkpointing protocols differ in the way they deal with consistent global states. Coordinated checkpointing protocols coordinate the processes at checkpoint time to ensure that the saved global state is consistent [11]. Communication-induced checkpointing provides the same guarantee without synchronizing the processes explicitly: they piggyback information on application messages instead [4]. Finally, uncoordinated checkpointing protocols take process checkpoints independently [5]. As a consequence, restoring the application in a consistent global state after a failure may lead to a cascade of rollbacks, known as *domino effect*.

Message logging protocols also apply to non deterministic applications. However, they consider a different execution model, called piecewise deterministic.

***Definition** 2 (Piecewise deterministic execution model):* An application execution is piecewise deterministic if all non deterministic events can be logged to be replayed identically after a failure.

Based on this execution model, an application can be recovered in the global consistent state observed before a failure if all non deterministic events that occurred before a failure were logged. Recovery can start from an inconsistent state and the missing events can be replayed from the logs. For the sake of simplicity, in the rest of the paper we consider the usual assumption in message logging protocols: the only non deterministic events are *recv* events.

Message logging protocols log *recv* events on a reliable storage in a determinant including the message identifier and its delivery order. Pessimistic or causal message logging protocols ensure that no determinant can be lost in a failure [1], and so provide perfect failure containment: Only the failed processes roll back after a failure.

A new execution model, called send-determinism, has been proposed recently to better qualify the execution of message passing HPC applications [10].

***Definition** 3 (Send-deterministic execution model):* An application execution is send-deterministic if, considering an initial state Σ^0, for each $p \in P \; and \; \forall S \in \mathscr{S}$, $S|p$ contains the same sub-sequence of *send* events.

In other words, the order of the *recv* events preceding a message sending has no impact on the sent message. Note that the send-deterministic execution model is *weaker*[2] than a deterministic execution model. A study of a large set of MPI HPC applications and benchmarks showed that this model holds for most of them [10]. Namely, in this study, Master/Worker applications are the only non send-deterministic applications.

Since the order of non causally dependent *recv* events does not impact the execution of the application, send-determinism allows to recover an application from an inconsistent global state without relying on event logging.This property was used to design a domino effect free uncoordinated checkpointing protocol [16].

C. Execution Model for an MPI Application

In an MPI application, the set of messages sent and received at the library level and at the application level can differ at some point during the execution. The relative order of the *send* and *recv* events might not even be the same at the two levels. One has to decide which events to consider when designing a rollback-recovery solution for MPI applications.

Figure 1 describes the set of events that can be associated with the reception of a message composed of several network packets. It describes a generic scenario where the application process posts an asynchronous reception request using *MPI_Irecv()* function, and then waits for its completion using *MPI_Wait()*. In the figure, events are outlined. Considering the MPI library level, a *lib_recv* and a *lib_complete* event are associated with the reception of the first and the last network packet respectively. At the application level, a *Request* event is set when the process posts a reception request to the library and a *Delivery* event is set when the message is delivered to the application. Two additional events can be defined. A *Matching* event is associated with the matching of the first message packet received at the MPI level to the posted request. Before delivering a message, the process has to check if the request is completed, *i.e.* if the whole message has been received. A *Completing* event is associated with the successful request completion checking.

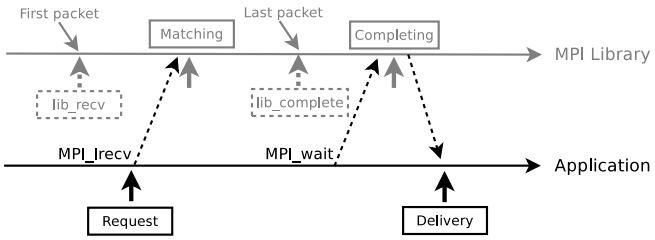

Figure 1. Events related to message reception on a MPI process

Figure 2 describes the set of events that can be associated with the sending of a message. Sending events are needed to

[2]Here, *weaker* means that it can apply to more applications.

define the *happened-before* relation between two messages. So the only events to take into account is the start of the message sending. At the application level, we consider the time when the request is posted to the library (*Post* event). At the library level, we consider the time when the first network packet is sent(*lib_send* event).

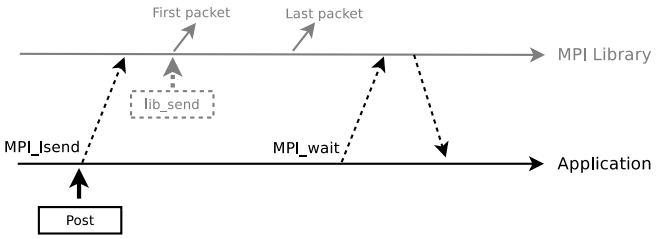

Figure 2. Events related to message sending on a MPI process

In [7], the authors propose to consider the application execution model from the library level. Doing so, they can reduce the number of non deterministic events in a piecewise deterministic execution model by taking into account the semantic of MPI. More precisely, they consider the *Matching* and *Completing* events in a message reception. In MPI, a *Matching* event is deterministic except if the source of the message is not specified in the reception request (use of the *MPI_ANY_SOURCE* wild card). Similarly, only some MPI completion functions are not deterministic, *e.g. MPI_Wait_any* or *MPI_Wait_some*. Reducing the number of non-deterministic events to log is important to improve message logging protocols performance [9].

In the send-deterministic model, we consider events at the application level. The relevant events to express causal dependencies are *Post* and *Delivery*. Note that by considering application level events, we do not put any constraint on the MPI library, *i.e.*, there is no requirement for the library to be send-deterministic. Using library level events could introduce *false happened-before* relations. Indeed, a *lib_complete* event can happen before a *lib_send* event, but the message has not been delivered to the application, it does not impact the following sent messages. Note that most HPC applications using *MPI_ANY_SOURCE* or non deterministic completion functions are send-deterministic since they are designed in such a way that message receptions order has no impact on the execution [10].

III. Failure Containment without Event Logging

We propose HydEE, a hybrid rollback-recovery protocol for send-deterministic applications. Hybrid rollback-recovery protocols have been proposed to combine the advantages of two existing protocols [31]. Application processes are divided into clusters, and a different protocol is used for the communications within a cluster (local level) and between clusters (global level). Combining coordinated checkpointing at the local level and message logging at the global level provides failure containment without logging all messages. Such protocols have been proposed in the piecewise deterministic execution model [32], [22], [8]. However, it has been proved that for such a protocol to be correct in this model, all the non deterministic events of the execution have to be logged reliably [8]. This includes the determinants of intra-cluster messages. HydEE leverages the send-deterministic model to combine coordinated checkpointing and message logging without logging any non deterministic event. Only the content of inter-cluster messages is logged in the sender memory [19].

Since HydEE does not log non deterministic events during the failure free execution, the application has to be recovered from an *inconsistent* global state after a failure. Replaying messages during recovery has to be done in the correct order with respect to orphan messages. To deal with this issue, the protocol assigns a *phase* number to all messages in the application, such that: a message has a higher phase than all inter-cluster messages it causally depends on. We show in Section IV that in a send-deterministic execution, replaying messages according to phase numbers after the failure is enough to ensure that a consistent global state is recovered.

This section provides a detailed description of HydEE. We first describe HydEE failure free protocol. Then we present the protocol on recovery, focusing on the use of phase numbers. To orchestrate recovery, an additional process, called *recovery process*, has to be launched when a failure occurs. Finally, we discuss garbage collection issues.

A. Providing Failure Containment

Algorithm 1 presents HydEE failure free protocol. It combines a coordinated checkpointing protocol inside the clusters with a message logging protocol between them. Sender-based message logging is used to log the messages content: each message payload is saved in the local memory of its sender (lines 7-8 of Algorithm 1). As mentioned before, message logging in HydEE is not combined with event logging. The messages logs are included in the checkpoints saved on reliable storage (lines 19-21 of Algorithm 1). Thus when a process rolls back, it looses the messages logged since the last checkpoint, but it is not an issue since those messages will be generated again during recovery.

In the event of a failure, only the processes belonging to the same cluster as the failed processes have to roll back. Inter-cluster messages are replayed from the logs. Only inter-cluster messages may become orphans since coordinated checkpointing protocols guarantee that the set of process checkpoints saved in a cluster is consistent. HydEE manages to avoid the rollback of the receiver of such messages without using any event logging because it leverages the send-deterministic assumption: the reception order of messages has no impact on message sending. To better illustrate how send-determinism is used, Figure 3 presents an execution with eight processes divided into three clusters. Figure 4

Algorithm 1 Failure Free Algorithm

Local Variables:

1: P_i, $Date_i$, $Phase_i$ {the ID, the date and the phase of the process i}
2: $Cluster_i$ {ID of the cluster the process i belongs to}
3: $Logs_i \leftarrow \emptyset$ {Set of messages logged by P_i}
4: $RPP_i \leftarrow [\perp, ..., \perp]$

5: **Sending message** msg **to** P_j
6: $Date_i \leftarrow Date_i + 1$
7: **if** $Cluster_i \neq Cluster_j$ **then**
8: $Logs_i \leftarrow Logs_i \cup (P_j, Date_i, Phase_i, msg)$
9: Send $(msg, Date_i, Phase_i)$ to process P_j

10: **Upon receiving** $(msg, Date_{send}, Phase_{send})$ **from** P_j
11: **if** $Cluster_j \neq Cluster_i$ **then**
12: $Phase_i \leftarrow Max(Phase_i, Phase_{send} + 1)$
13: $RPP_i[j].Maxdate \leftarrow Date_{send}$
14: $RPP_i[j][Date_{send}].phase \leftarrow Phase_{send}$
15: **else**
16: $Phase_i \leftarrow Max(Phase_i, Phase_{send})$
17: $Date_i \leftarrow Date_i + 1$
18: Deliver msg to the application

19: **Upon checkpoint in** $Cluster_i$
20: Coordinate with the processes in $Cluster_i$
21: Save $(ImagePs_i, RPP_i, Logs_i, Phase_i, Date_i)$ on stable storage

is a temporal representation of this execution. Grey squares represent checkpoints. If the processes of $Cluster_3$ fail and roll back, the message m_7 becomes an orphan message. Thanks to send-determinism, message m_5 or message m_6 can be received first, the same message m_7 will be sent anyway. Thus, the processes of $cluster_2$ do not need to roll back to receive m_7 again.

The rolled back processes may need messages from processes in other clusters to recover. But since these messages are logged, they are re-sent without requiring the rollback of their senders.

B. Recovery without Event Logging

Even if send-determinism ensures that the reception order of the messages has no impact on the application execution, messages are partially ordered by the *happened-before* relation. If the sending of a message m' depends on the reception of a message m, HydEE has to guarantee that, during recovery, m' will not be re-sent before m is received. In Figure 4, in a failure free execution, messages m_3 and m_7 cannot be sent before receiving m_1 and m_3 respectively. If a failure occurs in $Cluster_2$, all the processes of this cluster roll back to their last checkpoint and m_3 becomes an orphan message. The process p_4 can receive the logged message m_7 and send the message m_8 just after it restarts. The message m_8 may then be received before m_2. However, m_8 depends on m_2. The execution would not be correct. Two other scenarios can lead to the same problem: i) if message m_7 was not sent yet when the failure occured, it can be sent just after p_4 restarts; ii) if both $Cluster_2$ and $Cluster_3$ roll back, m_7 can be sent during recovery of $Cluster_3$.

The difference with a failure free execution lies in the existence of orphan messages: the processes of $Cluster_3$ do not need to receive m_3 to send or re-send the messages that depend on it. Processes should be able to know when a message they send may depend on an orphan message.

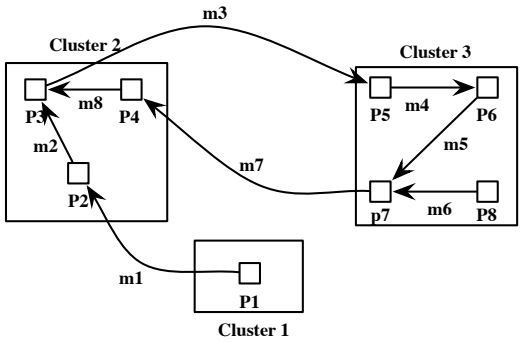

Figure 3. HydEE Execution Scenario with Three Clusters of Processes

As mentionned previously, to replay messages according to causal dependencies, we use phase numbers. This idea is adapted from a protocol we proposed previously [16]. A message m' depending on a message m that comes from another cluster should have a greater phase number than m. During recovery, m' will not be sent until all orphan messages with a lower phase than its own phase are received. Phases are implemented in the following way: Each process has a phase number that is piggybacked on each message it sends. When a process receives a message, it updates its own phase this way:

- If the message is an intra-cluster message, its phase becomes the maximum between its current phase and the one contained in the message (line 16 of Algorithm 1).
- If it is an inter-cluster message, its phase becomes the maximum between the message phase incremented by 1 and its current phase (line 12 of Algorithm 1).

The phase of a message is then always greater than the phase of the inter-cluster messages it depends on. To replay a message m in phase ρ, a process has to make sure that all orphan messages with a lower phase than ρ have been replayed.

To illustrate how phase numbers are used, we take the example of Figure 4. All processes phases are initialized to 1. When the process p_2 receives the message m_1, its phase becomes 2, the same for p_3 after receiving m_2. When p_5 receives m_3, its phase becomes 3 then the processes p_6 and p_7 update their phase to 3 after receiving m_4 and m_5. The phase of the message m_7 is then 3. After the failure and the rollback of the processes of $Cluster_2$, the message m_7 which phase is 3 cannot be sent until the orphan message m_3, which phase is 2 is received. Thus m_8 cannot be sent until m_3 is received.

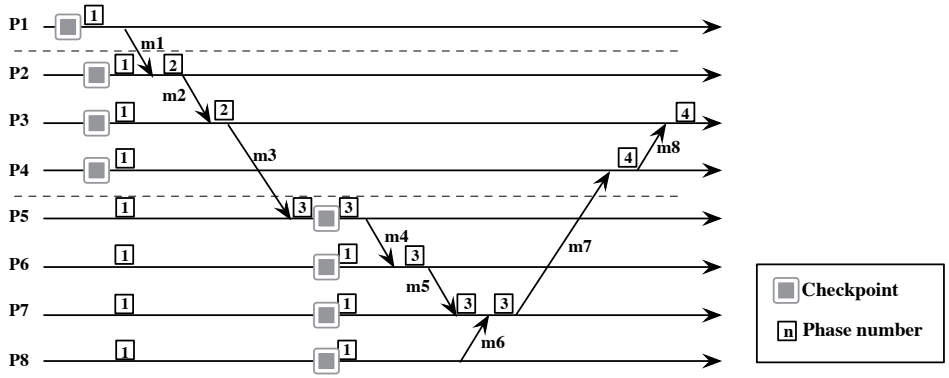

Figure 4. Temporal Representation of the Execution Scenario

C. Computing the Channels State

Algorithm 2 present the protocol executed by the processes that have to roll back after a failure. Algorithm 3 is the corresponding protocol for the processes that do not roll back. After a failure, the first step is to evaluate the state of the communication channels to compute the list of logged messages that have to be replayed as well as the list of orphan messages. Note that to uniquely identify *send* and *recv* events, each process has a date which is incremented after each event (lines 6 and 17 of Algorithm 1).

To compute channels state, each rolled back process sends a rollback notification that contains the date it restarts from to all the processes in the other clusters (line 6 of Algorithm 2). When a process p receives the rollback notification from a process q, it computes the set of logged messages it has to send q during recovery (lines 10-12 of Algorithm 3).

In order to compute the set of orphan messages, each process maintains a table called RPP (Received Per Phase). It contains as many entries as incoming channels for this process. For the channel from process p_j, it stores the date of the last received message ($RPP[j].Maxdate$) as well as the phase and the date of all received messages ($RPP[j][date].phase$) (line 13-14 of Algorithm 1). All the messages in $RPP[j]$ whose date is greater than the date of the rolled back process p_j are orphan messages (lines 13-14 of Algorithm 3).

Finally, non rolled back processes answer to the rolled back processes with a message containing the date of the last message received from them (line 9 of Algorithm 3). This information is used during recovery to know if a message to be sent is orphan (lines 14-15 of Algorithm 2).

D. Orchestrating Recovery

After a failure, a process called *recovery process* is launched (Algorithm 4). It ensures that a message cannot be sent as long as there are orphan messages in a lower phase. This rule applies to logged messages (lines 23-24 of Algorithm 3), to the first message sent by a process restarting from a checkpoint (line 8 of Algorithm 2) and also to the first message sent after a failure by the non-rolling back processes (line 18 of Algorithm 3). In the two latter cases, it is needed only for the first message because a process then waits until all orphan messages its state depends on are received. Thus, when it can send the first message, its state does not depend on any orphan message. In Figure 4, if both $Cluster_2$ and $Cluster_3$ roll back, the process p_7 is blocked waiting to be allowed to send m_7. After the orphan message m_3 is replayed, the process p_7 does not depend on any other

Algorithm 2 Rolled Back Processes Algorithm

Local Variables:
1: P_i, $Date_i$, $Phase_i$, $Cluster_i$
2: $Logs_i \leftarrow \emptyset$
3: $OrphanDate_i \leftarrow [\perp, ..., \perp]$ {$OrphanDate_i[j]$ is the date of the last orphan received by P_j}

4: **Upon failure of process** $P \in Cluster_i$
5: Restart from last ($ImagePs_i$, RPP_i, $Logs_i$, $Phase_i$, $Date_i$) on stable storage
6: Send($Rollback$, $Date_i$) to all $P_k \notin Cluster_i$
7: Send($OwnPhase$, $Phase_i$) to the recovery process
8: **wait until** receiving ($NotifySendMsg$, $Phase_i$) from the recovery process and receiving ($LastDate$, $date$) from all $P_k \notin Cluster_i$

9: **Upon receiving** ($LastDate$, $date$) **from** P_j
10: $OrphanDate_i[j] \leftarrow date$

11: **Sending message** msg **to** P_j
12: $Date_i \leftarrow Date_i + 1$
13: **if** $Cluster_i \neq Cluster_j$ **then**
14: **if** $Date_i \leq OrphanDate_i[P_j]$ **then**
15: Send ($OrphanNotification$, $Phase_i$) to the recovery process
16: **else**
17: Send (msg, $Date_i$, $Phase_i$) to process P_j
18: $Logs_i \leftarrow Logs_i \cup (P_j, Date_i, Phase_i, msg)$
19: **else**
20: Send (msg, $Date_i$, $Phase_i$) to process P_j
21: **if** $\forall P_k \notin Cluster_i$, $Date_i > OrphanDate_i[P_k]$ **then**
22: Switch back to failure free functions
23: **Upon Receiving message** (msg, $Date_{send}$, $Phase_{send}$) **from** P_j
24: Use failure free receive function
25: **Upon checkpoint in** $Cluster_i$
26: Use failure free checkpoint function

Algorithm 3 Non-Rolled Back Processes Algorithm

Local Variables:

1: P_i, $Date_i$, $Phase_i$, $Cluster_i$, RPP_i, $Logs_i$
2: $RollbackDate_i \leftarrow [\perp, ..., \perp]$ {$RollbackDate_i[j]$ is the date P_j rolls back to}
3: $ResentLogs_i \leftarrow \emptyset$ {List of logged messages to re-send}
4: $LogPhase_i \leftarrow \emptyset$ {Set of phases of logged messages}
5: $OrphPhases_i \leftarrow \emptyset$ {List including the phase of each orphan message}

6: **Upon failure of process** $P_j \notin Cluster_i$
7: **wait until** receiving ($Rollback$, $Date_{rb}$) from all $P_k \in Cluster_j$
8: **for all** $P_k \in Cluster_j$ **do**
9: Send($LastDate$, $RPP_i[k].MaxDate$) to p_k
10: **for all** $(P_k, date, phase, msg) \in Log_i$ **such that** $date > RollbackDate_i[k]$ **do**
11: Add $(P_k, date, phase, msg)$ to $ResentLogs_i$
12: $LogPhase_i \leftarrow LogPhase_i \cup phase$
13: **for all** $date \in RPP_i[k]$ **such that** $date > RollbackDate_i[k]$ **do**
14: Add $RPP_i[k][date].phase$ to $OrphPhases_i$
15: Send(Log, $LogPhase_i$) to the recovery process
16: Send($Orphan$, $OrphPhases_i$) to the recovery process
17: Send($OwnPhase$, $Phase_i$) to the recovery process
18: **wait until** receiving ($NotifySendMsg$, $Phase_i$) from the recovery process
19: Use failure free functions

20: **Upon receiving** ($Rollback$, $Date_{rb}$) **from** P_j
21: $RollbackDate[P_j] \leftarrow Date_{rb}$

22: **Upon receiving** ($NotifySendLog$, $Phase_{notif}$) **from the recovery process**
23: **for all** $(P, date, phase, msg) \in ResentLogs_i$ **such that** $Phase \leq Phase_{notif}$ **do**
24: Send (msg, $Date$, $Phase$, $Cluster_i$) to P

Algorithm 4 Algorithm for the Recovery Process

Local Variables:

1: $NbOrphPhase \leftarrow \emptyset$ {$NbOrphPhase[\rho]$ is the number of processes waiting for an orphan message in phase ρ}
2: $ProcessPhases \leftarrow \emptyset$ {$ProcessPhases[\rho]$ is the set of processes in phase ρ}
3: $MsgLPhase \leftarrow [\perp, ..., \perp]$ {$MsgLPhase[\rho]$ is the set of processes that have at least one logged message in phase ρ}

4: **Upon receiving** (Log, $LogPhase$) **from process** P_j
5: **for all** $phase \in LogPhase$ **do**
6: $MsgLPhase[phase] \leftarrow MsgLPhase[phase] \cup P_j$

7: **Upon receiving** ($Orphan$, $OrphPhases$) **from process** P_j
8: **for all** $phase \in OrphPhases$ **do**
9: $NbOrphanPhase[phase] \leftarrow NbOrphanPhase[phase] + 1$

10: **Upon receiving** ($OwnPhase$, $phase$) **from** P_j
11: $ProcessPhase[phase_j] \leftarrow ProcessPhase[phase] \cup Pj$

12: **Upon receiving** ($OrphanNotification$, $phase$) **from** P_j
13: $NbOrphanPhase[phase] \leftarrow NbOrphanPhase[phase] - 1$
14: **if** $NbOrphanPhase[phase] == 0$ **then**
15: Start **NotifyPhase**

16: **NotifyPhase**
17: **for all** $phase \in MsgLPhase$ **such that** $\nexists phase' < phase \wedge NbOrphanPhase[phase'] > 0$ **do**
18: **for all** $P_k \in MsgLPhase[phase]$ **do**
19: Send ($NotifySendLog$, $phase$) to P_k
20: Remove $phase$ from $MsgLPhase$
21: **for all** $phase \in ProcessPhase$ **such that** $\nexists phase' < phase \wedge NbOrphanPhase[phase'] > 0$ **do**
22: **for all** $P_k \in ProcessPhase[phase]$ **do**
23: Send ($NotifySendMsg$, $phase$) to P_k
24: Remove $phase$ from $ProcessPhase$

orphan message, and so, it is allowed to send messages.

When a failure occurs, the rolled back processes send to the recovery process the phase they restart from, *i.e.*, the one contained in the checkpoint (line 7 of Algorithm 2). The other processes send the phase of the logged messages they should send (lines 10-15 of Algorithm 3) and the phase of the orphan messages they should receive (lines 13-16 of Algorithm 3). The recovery process uses the first information to know which processes to notify in each phase and the second one to know how many orphans there are in each phase. Finally, since a process in phase ρ should not send any new message until there are no orphan messages in a lower phase than ρ, it sends its current phase to the recovery process (lines 18-17 of Algorithm 3).

Each time a rolled back process has to send an orphan message, it sends a notification to the recovery process instead of sending the real message (lines 14-15 of Algorithm 2) since send-determinism ensures that the message would be the same as the one sent before the failure. When all notifications for orphans messages in one phase have been received (lines 14-15 of Algorithm 4), the $NotifySendLog$ (notification for logged messages) and $NotifySendMsg$ (notification for non-logged messages) notifications for the next phases with no orphans are sent (lines 17-20 and lines 21-23 of Algorithm 4).

If the recovery process fails during recovery, another one could be started to replace it. It would just need to synchronize again with the application processes to know the orphan messages and logged messages to replay that remain in the current application state.

E. Garbage collection

In a coordinated checkpointing protocol, only the last checkpoint is needed. So, in our case, logged messages received before the last checkpoint are not needed anymore. To delete these messages, after a checkpoint, each process answers with an acknowledgment, containing its current date d, to the first message received from each process from another cluster. When a process receives the acknowledgment, it deletes all the messages for this process and all the RPP entries with a date lower than d.

IV. Proof of correctness

As defined in Section II-A, a correct execution for a parallel application is an execution where:

- The sequence of events S is consistent with the *happened-before* relation.
- The global state at the end of the execution is final.

We prove that HydEE ensures a correct application execution despite multiple concurrent failures. We first provide lemmas defining some characteristics of our protocol. Then we show that despite failures, HydEE ensures that the sequence of events during the execution is correct with respect to the *happened-before* relation. Finally, to show that the execution will reach a final state, we prove that our protocol is deadlock free.

For the sake of simplicity, we only consider $send$ and $recv$ events. We denote by $Ph(x)$ the phase of x, x being an event or a process state. If event $e \in S|p_i$ results in state σ_i^k, then $Ph(e) = Ph(\sigma_i^k)$.

A. Protocol Characteristics

Lemma 1: Let e and e' be two events with $e \rightarrow e'$, then $Ph(e) \leq Ph(e')$.

Proof: Consider a causal chain connecting e to e': $e = e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_i = e'$. The proof is by induction on the length of this causal chain.

- *Base case* $i = 1$: We distinguish two cases:
  1) Events e and e' are on the same process P_i: e, e' can be $send$ or $recv$ events. If event $e_i^k = send(m)$, $Ph(\sigma_i^{k-1}) = Ph(\sigma_i^k)$. If event $e_i^k = recv(m)$, $Ph(\sigma_i^{k-1}) \leq Ph(\sigma_i^k)$ (Lines 11 and 16 of Algorithm 1). So $Ph(e) \leq Ph(e')$.
  2) Events e and e' are on different processes: in this case, $e = send(m)$ and $e' = recv(m)$, and so, $Ph(e) \leq Ph(e')$ (Lines 11 and 16 of Algorithm 1).
- *Induction step*: Consider a causal chain of length $i+1$: $e = e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_i \rightarrow e_{i+1}$. By induction hypothesis, the result holds for a causal chain of length i, $Ph(e_0) \leq Ph(e_i)$. Consider $e_i \rightarrow e_{i+1}$. By the same reasoning as in the base case, we have $Ph(e_i) \leq Ph(e_{i+1})$. Together, we have $Ph(e_0) \leq Ph(e_{i+1})$, which concludes the induction step.

■

Lemma 2: If m is an orphan message then m is an inter-cluster message.

Proof: Since coordinated checkpointing protocols guarantee that the set of process checkpoints saved in a cluster is consistent, orphan messages are inter-cluster messages. ■

Lemma 3: Let m and m' be two messages such that $recv(m) \rightarrow send(m')$. If m is an orphan message then $Ph(m) < Ph(m')$.

Proof: Consider a causal chain connecting $recv(m)$ to $send(m')$: $recv(m) = e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_i = send(m')$. Since orphan messages are inter-cluster messages (Lemma 2), $Ph(e_0) < Ph(e_1)$ (Line 11-12 of Algorithm 1). Since $Ph(e_1) \leq Ph(e_i)$, $Ph(e_0) < Ph(e_i)$ ■

Lemma 4: Let e be an event with $e = send(m)$. In any correct execution, e has the same phase.

Proof: We assume $send(m)$ is event e_i^k on process p_i. $Ph(send(m))$ only depends on the messages received by p_i before $send(m)$ (Lines 11 and 16 of Algorithm 1). Since we consider the send deterministic execution model, the same messages are sent in any correct execution (Definition 3). Recall that $\mathscr{S}$ is the set of correct executions: for each $p \in P$ *and* $\forall S \in \mathscr{S}$, $S|p$ contains the same sub-sequence of $send$ events. It implies that for each $p \in P$ *and* $\forall S \in \mathscr{S}$, $S|p$ contains the same set of $recv$ events. Since events on a process are ordered by the *happened-before* relation, $S|p_i^k$ contains the same set of $recv$ events $\forall S \in \mathscr{S}$. Thus $Ph(send(m))$ is the same in any correct execution. ■

B. Consistency of the Sequence of Events

Let Σ_r be the set of processes states after the failure of some processes (in the same or different clusters), *i.e.* after the rollbacks. We say that event $e_i^k = send(m)$ on process p_i can happen in state Σ, if the state of process p_i in Σ is σ_i^{k-1}. To prove that after a failure, the sequence of events in the application is consistent with the *happened-before* relation, we prove that if $\exists send(m)$ *and* $send(m')$, with $send(m) \rightarrow send(m')$, that can both happen after a rollback, HydEE ensures that $send(m')$ cannot happen until $send(m)$ happens.

Lemma 5: Consider events $send(m)$ and $send(m')$, with $send(m) \rightarrow send(m')$, that can both happen in state Σ_r. Then $\exists m''$, that can be the message m, such that $send(m) \rightarrow send(m'') \rightarrow send(m')$ and m'' is an orphan.

Proof: We prove that by contradiction. Recall that m'' is orphan in global state Σ if $recv(m'') \in \Sigma$ but $send(m'') \notin \Sigma$. Consider a causal chain connecting $send(m)$ to $send(m')$: $send(m) = send(m_1) \rightarrow recv(m_1) \rightarrow send(m_2) \ldots \rightarrow recv(m_{n-1}) \rightarrow send(m_n) = send(m')$. Since $send(m')$ can happen, $recv(m_{n-1}) \in \Sigma_r$. Since message m_{n-1} is not orphan, $send(m_{n-1}) \in \Sigma_r$ too. We apply the same reasoning for all messages in the chain until: $recv(m_1) \in \Sigma_r$ implies $send(m_1) \in \Sigma_r$, which is a contradiction since $send(m)$ can happen. ■

Theorem 1: If $\exists send(m)$ *and* $send(m')$, with $send(m) \rightarrow send(m')$, that can both happen, $send(m')$ cannot happen until $send(m)$ occurs.

Proof: Since $send(m) \rightarrow send(m')$, and both can happen, $\exists m''$ such that $send(m) \rightarrow send(m'') \rightarrow send(m')$ and m'' is an orphan (Lemma 5). Thus $Ph(send(m)) \leq Ph(send(m'')) < Ph(send(m'))$. According to line 8 of Algorithm 2, and lines 18 and 22 of Algorithm 3, a $send$ event cannot occur while there are orphans in a lower phase. So $send(m')$ cannot occur before $send(m'')$ occurs. Without loss of generality, we assume that m'' is the only orphan event in the causality chain. Since $send(m) \rightarrow send(m'')$, $send(m'')$ cannot occur before $send(m)$ occurs. So, $send(m')$ cannot occur before $send(m)$ occurs. ■

C. Deadlock Free Recovery

Deadlocks in HydEE could occur if some notifications for one phase are not received by the recovery process, and

so it could not notify processes to send some messages. Notifications are sent to the recovery process when $send(m)$ occurs, with m orphan (line 15 of Algorithm 2). It cannot happen that a notification is sent with an incorrect phase number because of Lemma 4. So we just have to prove that during recovery, all orphan messages are eventually re-sent.

***Theorem** 2:* During recovery, $\forall m$ such that m is an orphan message, $send(m)$ eventually occurs.

Proof: Let $\mathscr{O}$ be the set of orphan messages in state Σ_r. Let min_phase be the smallest $Ph(send(m))$, $\forall m \in \mathscr{O}$. According to lines 17-20 and 21-23 of Algorithm 4, all $send$ events such that $Ph(send) \leq min\_phase$ can occur. According to Lemma 1, if $send(m) \rightarrow send(m')$, then $Ph(send(m)) \leq Ph(send(m'))$. So all messages $m \in \mathscr{O}$ with $Ph(send(m)) = min\_phase$ can eventually be replayed. Then the same reasoning applies to the new min_phase, until $\mathscr{O} = \emptyset$. ■

V. Evaluation

We implemented HydEE in the MPICH2 library[3]. In this section, we present our experimental results. First, we describe our prototype and our experimental setup. Then we present the performance evaluation of HydEE on failure free execution using NetPIPE [30] and the NAS Parallel Benchmark Suite [3].

A. Prototype Description

We integrated HydEE in the nemesis communication subsystem of MPICH2. HydEE works for TCP and Myrinet/MX channels. We focus on the Myrinet/MX implementation.

The main modification we applied to the communication system is related to the phase number and the date that have to be sent along with every application message. To implement an efficient data piggybacking mechanism, we use two different solutions, based on the size of the application message. In MX, data can be added to the application message simply by adding one more segment to the list of segments passed to the *mx_isend()* function. However sending non-contiguous buffers in the same message can result in extra memory copies. Thus we use this solution only to optimize latency for small messages (below 1 Kilo-Byte). For large messages (over 1 Kilo-Byte), we send the protocol data in a separate message to avoid any extra memory copy that would impair communication performance.

To implement sender-based message logging, we simply copy the content of the messages in a pre-allocated buffer using *memcpy* libc call. The study presented in [6] shows that it is theoretically possible to implement sender-based message logging without any extra cost because the latency and the bandwidth provided by *memcpy* are better than the one provided by Myrinet 10G. In our implementation, the message payload copy is done between the *mx_isend()* call and the corresponding *mx_wait()* request completion call, to overlap in-memory message copy and message transmission on the network.

[3]https://svn.mcs.anl.gov/repos/mpi/mpich2/trunk:r7375

B. Experimental Setup

1) Testbed: We run our experiments on Lille cluster of Grid'5000. We use 41 nodes equipped with 2 Intel Xeon E5440 QC (4 cores) processors, 8 GB of memory , and 25 nodes equipped with 2 AMD Opteron 285 (2 cores) processors, 4 GB of memory. All nodes are equipped with a 10G-PCIE-8A-C Myri-10G NIC. Operating system is Linux (kernel 2.6.26).

2) Applications Description: Our evaluation includes two tests. First, we evaluate the impact of HydEE on communication performance using NetPIPE. NetPIPE is a ping-pong test used to measure latency and bandwidth between two nodes. Second, we evaluate the impact of HydEE on applications failure free performance. For this, we use 6 class D NAS benchmarks running on 256 processes.

3) Applications Process Clustering: To run an application with HydEE, clusters of processes have to be defined. To do so, we use the tool described in [28]. It tries to find a clustering configuration that provides a good trade-off between size of the clusters and amount of communications to log. It takes as input a graph defining the amount of data sent in each application channel. To get the communication pattern of the applications, we modified MPICH2 to collect data on communications.

Table I presents the clustering configuration we use in our experiments. The table includes the number of clusters, the percentage of the processes that would roll back in the event of a failure assuming that failures are evenly distributed over all processes, and the ratio of logged data. For all applications except FT, the clustering configuration ensures that less or around 20% of the processes would roll back after a failure while logging less than 20% of the messages. FT does not provide such good results because of the use of all-to-all communication primitives. Note that the results presented in [28] for the same applications run over 1024 processes show a better trade-off between clusters size and amount of data logged: less than 15% of processes to roll back with the same amount of logged data.

	Nb Clusters	Avg Ratio of Process to Roll Back (Single Failure Case)	Log/Total Amount of data (in GB)
NAS BT	5	21.78%	143/791 (18.09%)
NAS CG	16	6.25%	440/2318 (18.98%)
NAS FT	2	50%	431/860 (50.19%)
NAS LU	8	12.5%	44/337 (13.26%)
NAS MG	4	25%	13/66 (19.63%)
NAS SP	6	18.56%	289/1446 (20.04%)

Table I
Application clustering on 256 processes

C. Communication Performance

Figure 5 compares MPICH2 native communications performance over Myrinet 10G, to the performance provided by HydEE for two processes in the same cluster (without logging), and for two processes belonging to different clusters (with logging), using Netpipe [30]. The figure shows the performance degradation in percent for latency and bandwidth compared to the native performance of MPICH2. Results show, first, that HydEE induces a small overhead on communication performance, and only for small-sized messages. The two peaks in the performance degradation are due to the data piggybacked on messages. The reason is that there are plateau in the native performance of MPICH2 over MX. For instance, in our experiment, the native latency of MPICH2 is around 3.3 μs for messages size 1 to 32 bytes and then jump to 4 μs. Because of the message size increase due to the additional data sent, HydEE reaches these plateau earlier. Second, the performance with and without logging are equivalent. It means that our sender-based message logging technique has no impact on performance and that the overhead is only due to piggybacking.

One could argue that we should not use memory for message logging, since the application might need all the nodes memory. In this case, a solution based on additional local storage devices with good bandwidth performance, *e.g.* solid state disk, could be designed. A memory buffer would be used to copy the messages at the time they are sent, and a dedicated thread would copy the data from the memory buffer to the storage device asynchronously. This is part of our future work.

D. Applications Performance

Figure 6 presents an evaluation of HydEE failure free performance using the NAS benchmarks. It compares the performance of HydEE with process clustering to the native performance with MPICH2. The case where all application messages are logged is also evaluated. Results are mean values over 8 executions of each application and are presented as normalized execution time. The execution time with MPICH2 is chosen as reference. The results show that even in the cases where logging all messages content could induce a small overhead on the execution time, HydEE provides performances almost equivalent to MPICH2 native performance: overhead is at most 1.25%. Using partial message logging to reduce the amount of messages to log is beneficial for failure free performance, compared to full message logging. Since our algorithm does not rely on any central point during failure free execution, we can assume that these results would remain valid at very large scale.

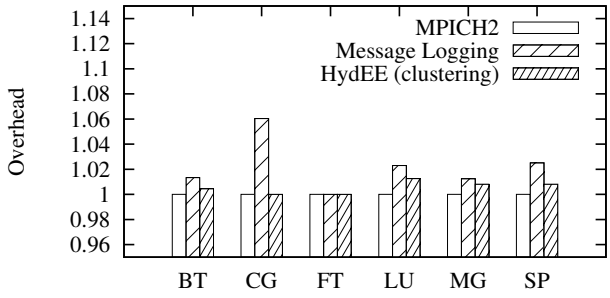

Figure 6. NAS Benchmark Performance over MX

VI. Related Work

As described in Section II, checkpointing protocols do not provide failure containment. At small scale, coordinated checkpointing is the solution of choice for failure free performance because it does not require to log any message or to piggyback any data. However, at large scale, saving all checkpoints at the same time may create an I/O burst that could slowdown the application [25]. Communication induced checkpointing has the same drawback since evaluations show that, with these protocols, the number of forced checkpoints is very high [2]. Using an uncoordinated checkpointing protocol allows to schedule checkpoints to avoid I/O bursts. An uncoordinated checkpointing protocol without domino effect for send-deterministic applications has been proposed [16]. However this solution does not provide failure containment.

Message logging protocols provide failure containment but they lead to a large memory occupation. Moreover, saving determinants on stable storage has a significant impact on communication performance [29]. As mention in Section II, an execution model for MPI applications can be defined to reduce the number of determinants to log [9]. However no evaluation have been conducted at very large scale yet to show if this optimization is efficient enough.

Hybrid protocols have been proposed to overcome the limits of the protocols described above. We do not discuss the protocols described in [24], [17], [31] since they fail in confining failures to one cluster.

All existing hybrid protocols assume a piecewise deterministic execution model. They use coordinated checkpoints inside clusters to get good failure free performance. Since these protocols allow checkpoint scheduling between the clusters, they can avoid checkpoints I/O bursts. As in HydEE, the protocols described in [8], [22], [32] use message logging between clusters to ensure that a failure in one cluster affects only this clusters. In these protocols, the determinants of all messages have to be logged reliably [8]. They differ in the way they handle determinants inside a cluster. In [32], a causal approach is used: determinants are piggybacked on messages until they are saved reliably. In order to avoid data piggybacking, the protocols described in [8] and [22] save determinants synchronously on stable storage. We proved that HydEE can handle multiple failures without logging any determinant.

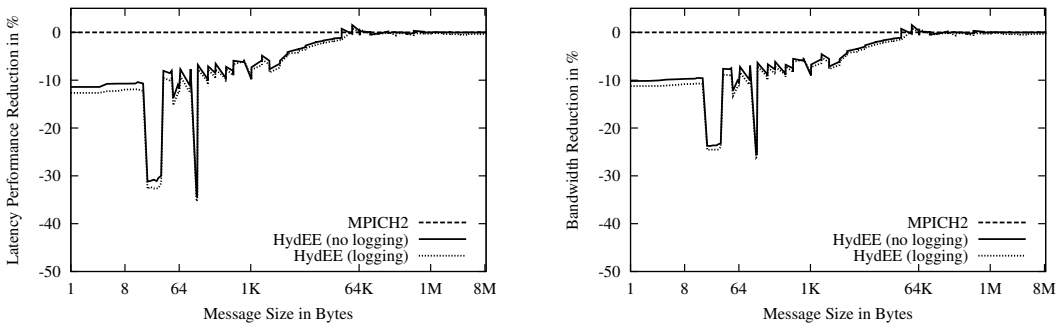

Figure 5. Myrinet 10G Ping-Pong Performance

Finally, the hybrid protocol proposed in [18] does not log any determinant. Thus, it can only work for deterministic applications, since the authors do not provide any solution to handle orphan messages. HydEE does not require the application to be deterministic.

VII. Conclusion

HydEE combines the advantages of coordinated checkpointing and message logging protocols in a hybrid rollback-recovery protocol for message passing applications. Application processes are clustered to apply coordinated checkpointing inside each cluster. Inter-cluster messages are logged to avoid rollback propagation. HydEE provides failure containment while logging only a subset of the application messages. Leveraging the send-deterministic execution model, it provides a unique feature compared to similar hybrid rollback-recovery protocols: it does not require to log any event on reliable storage to provide failure containment. HydEE is proved to be able to tolerate multiple concurrent failures. We implemented HydEE in the MPICH2 library. Experiments run on a high performance network with a set of benchmarks show that HydEE induces almost no overhead on failure free execution. Additionally, it shows that partial message logging is beneficial for failure free performance. All these properties make HydEE a promising candidate for extreme-scale fault tolerance.

As a future work, we will investigate solutions to cluster the application processes while the application is running, instead of using an off-line analysis. To be able to handle applications with a communication pattern that is evolving over time, the ability to handle dynamic clustering should be added to HydEE. We also plan to study how our protocol could be integrated with topology-aware multi-level checkpointing techniques [15].

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr). This work was supported by INRIA-Illinois Joint Laboratory for Petascale Computing and the ANR RESCUE and G8 ECS (Toward Exascale Climate Simulations) projects.

References

[1] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[2] L. Alvisi, S. Rao, S. A. Husain, A. de Mel, and E. Elnozahy. An Analysis of Communication-Induced Checkpointing. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 242–, Washington, DC, USA, 1999. IEEE Computer Society.

[3] D. Bailey, T. Harris, W. Saphir, R. van der Wilngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[4] R. Baldoni, F. Quaglia, and B. Ciciani. A VP-Accordant Checkpointing Protocol Preventing Useless Checkpoints. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, SRDS '98, pages 61–, Washington, DC, USA, 1998. IEEE Computer Society.

[5] B. Bhargava and L. Shu-Renn. Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems-an Optimistic Approach. In *Seventh Symposium on Reliable Distributed Systems*, pages 3–12, Columbus, OH , USA, 1988.

[6] G. Bosilca, A. Bouteiller, T. Herault, P. Lemarinier, and J. J. Dongarra. Dodging the Cost of Unavoidable Memory Copies in Message Logging Protocols. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 189–197, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] A. Bouteiller, G. Bosilca, and J. Dongarra. Redesigning the Message Logging Model for High Performance. *Concurrency and Computation : Practice and Experience*, 22:2196–2211, November 2010.

[8] A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Correlated Set Coordination in Fault Tolerant Message Logging Protocols. In *Euro-Par 2011*, volume 6853 of *Lecture Notes in Computer Science*, pages 51–64. Springer Berlin / Heidelberg, 2011.

[9] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra. Reasons for a Pessimistic or Optimistic Message Logging Protocol in MPI Uncoordinated Failure Recovery. In *IEEE International Conference on Cluster Computing (Cluster 2009)*, New Orleans, USA, 2009.

[10] F. Cappello, A. Guermouche, and M. Snir. On Communication Determinism in Parallel HPC Applications. In *19th International Conference on Computer Communications and Networks (ICCCN 2010)*, 2010.

[11] K. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[12] J. Dongarra, P. Beckman, T. Moore, et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25:3–60, February 2011.

[13] E. N. Elnozahy et al. System Resilience at Extreme Scale. Technical report, DARPA, 2008.

[14] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[15] L. B. Gomez, N. Maruyama, D. Komatitsch, S. Tsuboi, F. Cappello, S. Matsuoka, and T. Nakamura. FTI: high performance Fault Tolerance Interface for hybrid systems. In *IEEE/ACM SuperComputing 2011*, Seatle, USA, November 2011.

[16] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic Message Passing Applications. In *25th IEEE International Parallel & Distributed Processing Symposium (IPDPS2011)*, Anchorage, USA, 2011. to appear.

[17] B. Gupta, R. Nikolaev, and R. Chirra. A Recovery Scheme for Cluster Federations Using Sender-based Message Logging. volume 19, pages 127–139, 2011.

[18] J. C. Y. Ho, C.-L. Wang, and F. C. M. Lau. Scalable Group-Based Checkpoint/Restart for Large-Scale Message-Passing Systems. In *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, USA, 2008.

[19] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Digest of Papers: The 17th Annual International Symposium on Fault-Tolerant Computing*, pages 14–19, 1987.

[20] P. Kogge et al. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, DARPA, 2008.

[21] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[22] E. Meneses, C. L. Mendes, and L. V. Kale. Team-based Message Logging: Preliminary Results. In *3rd Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids (CCGRID 2010).*, May 2010.

[23] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. www.mpi-forum.org, 1995.

[24] S. Monnet, C. Morin, and R. Badrinath. Hybrid Checkpointing for Parallel Applications in Cluster Federations. In *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid (CCGRID'04)*, pages 773–782, Washington, DC, USA, 2004. IEEE Computer Society.

[25] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the Impact of Checkpoints on Next-Generation Systems. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, Washington, DC, USA, 2007. IEEE Computer Society.

[26] S. Rao, L. Alvisi, and H. M. Vin. The Cost of Recovery in Message Logging Protocols. In *Symposium on Reliable Distributed Systems*, pages 10–18, 1998.

[27] R. Riesen, K. Ferreira, and J. Stearley. See Applications Run and Throughput Jump: The Case for Redundant Computing in HPC. In *Proceedings of the 2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*, DSNW '10, pages 29–34, Washington, DC, USA, 2010. IEEE Computer Society.

[28] T. Ropars, A. Guermouche, B. Uçar, E. Meneses, L. V. Kalé, and F. Cappello. On the Use of Cluster-Based Partial Message Logging to Improve Fault Tolerance for MPI HPC Applications. In *Euro-Par 2011*, pages 567–578, 2011.

[29] T. Ropars and C. Morin. Active optimistic and distributed message logging for message-passing applications. *Concurrency and Computation: Practice and Experience*, 23(17):2167–2178, 2011.

[30] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, June 1996.

[31] N. Vaidya. Distributed recovery units: An approach for hybrid and adaptive distributed recovery. *Technical Report 93-052, Departement of Computer Science Texas, A&M, University*, 1993.

[32] J.-M. Yang, K. F. Li, W.-W. Li, and D.-F. Zhang. Trading Off Logging Overhead and Coordinating Overhead to Achieve Efficient Rollback Recovery. *Concurrency and Computation : Practice and Experience*, 21:819–853, April 2009.