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

Amina Guermouche*, Thomas Ropars†, Marc Snir‡, Franck Cappello*‡

*INRIA Saclay-Ile 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, 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). By combining coordinated checkpointing at the local level with a message logging protocol at the global level [32], [22], [8], the consequences of a failure can be limited to a single cluster by logging only inter-cluster messages. 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 address the problem of 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.
- 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 and efficiency 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.

A. Message Passing System Model

To model a message passing parallel execution, we consider a set \( P = \{p_1, p_2, \ldots, p_n\} \) of \( n \) processes, and a set \( C \) of channels connecting any ordered pair of processes. Channels are assumed to be FIFO and reliable but we do not make any assumption on system synchrony.

An execution \( E \) is defined by an initial state \( \Sigma^0 = \{\sigma^0_1, \sigma^0_2, \ldots, \sigma^0_n\} \), where \( \sigma^0_i \) is the initial state of process \( p_i \), and a sequence \( S = e_1, e_2, e_3, \ldots \) 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 \( \sigma_i^k \).

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 \( \sigma_i^k \) of process \( p_i \) can be defined as \( \sigma_i^k = (\sigma_i^0, S[p_i]^k) \), where \( S[p_i]^k = e_1^i, e_2^i, \ldots, e_l^i \). Checkpointing process \( p_i \) consists in saving a state \( \sigma_i \) on a reliable storage. A global state \( \Sigma \) is composed of one state of each process in \( P, \Sigma = \{\sigma^1, \sigma^2, \ldots, \sigma^n\} \). Event \( e_i^k \in \Sigma \) if \( \sigma_i^k \in \Sigma \) and \( k \leq l \).

A process state \( \sigma_i^k \) is final if no transition is possible to \( \sigma_i^{k+1} \). A global state \( \Sigma = \{\sigma_1^1, \sigma_2^2, \ldots, \sigma_n^n\} \) is final if all \( \sigma_i^k \) 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 \( E = (\Sigma^0, S) \) 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 \( \Sigma \) is consistent if for all events \( e, e' \):

\[
e' \in \Sigma \text{ and } e \rightarrow e' \implies e \in \Sigma
\]  

A message is said orphan in the global state \( \Sigma \) 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 additionally includes a specification of the application determinism. To reason about the determinism of an application, we need to consider \( \delta \), the set of correct executions from an initial application state \( \Sigma^0 \). The set \( \mathcal{F} \) includes the sequences of events \( S \) corresponding to the executions in \( \delta \).
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 \(\Sigma_0\), \(\exists S \in \mathcal{S} \) and \(S' \in \mathcal{S} \) and a process \(p \in P\) such that:

\[
S|p \neq S'|p
\]

(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 the 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 use the traditional assumption in message logging protocols: the only non deterministic events are \(recv\) events.

Message logging protocols log \(recv\) events on a reliable storage as a deterministic composed of the message identifier and its delivery order. Pessimistic or causal message logging protocols ensure that no determinist 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 applications [10].

**Definition 3 (Send-deterministic execution model):** An application execution is send-deterministic if, considering an initial state \(\Sigma_0\), for each \(p \in P\) and \(\forall S \in \mathcal{S}\), \(S|p\) 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 than a deterministic execution model.

The study of a large set of MPI HPC applications and benchmarks showed that this model holds for most of them [10]. 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. We used this property to design a domino effect free uncoordinated checkpointing protocol [16].

**C. Execution Model for a 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 \(MPI_{recv}()\) 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](image)

2Here, weaker means that it can apply to more applications.
Figure 2 describes the set of events that can be associated with the sending of a message. Sending events are useful to define the happened-before relation between two messages. So the only events to take into account correspond to 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.

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 wildcard). 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 events at the application level, we do not put any constraint on the MPI library, i.e. there is no requirement for the library to be send-deterministic. Furthermore, using library level events can introduce false happened-before relations in the sense that lib_complete event could happen before a lib_send, but as long as the message has not been delivered to the application, it does not actually impact the following sent messages. Note that both the use of MPI_ANY_SOURCE and non deterministic completion functions can lead to a send-deterministic execution.

III. FAILURE CONTAINMENT WITHOUT EVENT LOGGING

We propose HydEE, an 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 two clusters (global). Combining coordinated checkpointing at the local level and message logging at the global level is a good solution to provide failure containment without logging all messages. Such protocols have been designed 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 non deterministic events of the execution have to be logged reliably [8]. This include the determinants of intra-cluster messages. HydEE leverages the send-deterministic model to combine coordinated checkpointing and message logging without logging any non deterministic events. Only the content of inter-cluster messages is logged in the sender memory [19]: HydEE provides partial message logging.

In this section, we provide a detailed description of HydEE focusing on failure management. We first describe how failure containment is ensured. Then we describe how causal dependencies can be managed during recovery without relying on event logging. To do so, we introduce an additional process, called recovery process, which is launched when a failure occurs to orchestrate the application recovery.

A. Providing Failure Containment

Algorithm 1 presents the algorithm of processes in a failure free execution. HydEE combines a coordinated checkpointing protocol inside the clusters with a message logging protocol between them. A sender-based message logging protocol is used to log the message 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 any event logging. The messages logs are included in the checkpoints saved on reliable storage (lines 19-21 of Algorithm 1).

HydEE provides failure containment. In the event of a failure, only the processes belonging to the same cluster as the failed processes have to rollback. It relies on send-determinism to avoid rollbacks induced by orphan messages, and on message logging to replay inter-cluster messages.

In the event of a failure, 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 presents the temporal representation of the execution described in Figure 3. Grey squares represent the checkpoints. If the processes of Cluster3 fail and roll back, the message m5 becomes an orphan message. Thanks to send-determinism, this message will be re-sent regardless of the reception order of the messages m5 and
m8. Thus, the processes of cluster_2 do not need to roll back to receive m7 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 m3 and m7 cannot be sent before receiving m1 and m3 respectively. In the event of the failure of a process from Cluster_2, all the processes of this cluster roll back to their last checkpoint and m3 becomes an orphan message. The process p4 can receive the logged message m7 and send the message m8 just after it restarts. The message m8 may then be received before m2. However, m8 depends on m2. The execution would not be correct. Two other scenarios can lead to the same problem: i) if m7 is not sent from the logs (it was not sent before the failure of Cluster_2 processes), it can be sent just after p4 restarts; ii) if both Cluster_2 and Cluster_3 roll back, m7 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 m7 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.

To ensure causal delivery order during recovery, we adapt the idea we proposed in [16]: using phase numbers to indicate that a message depends on a possible orphan message. 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 be able to send a message m after a failure, a process p in a phase p has to make sure that all orphan messages with a lower phase than m phase are received.

To illustrate how phase numbers are used, we take the example of Figure 4. All processes phases are initialized to 1. When the process p2 receives the message m1 (which phase number is 1), its phase becomes 2 and p3 does the same after receiving m2. When p5 receives m3, its phase becomes 3 then the processes p6 and p7 update their phase to 3 after receiving m4 and m5. The phase of the message m7 is then 3. After the failure and the rollback of the processes of Cluster_2, the message m7 which phase is 3 cannot be sent until the orphan message m3, which phase is 2 is received. Thus m8 cannot be sent until m3 is received.

**C. Computing the Channels State**

Algorithms 2 and 3 describe the algorithms of rolled back and non rolled back processes in the event of a failure.
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, i.e., the set of orphan and logged messages that will be replayed on the channels, 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 on the channel and their phases (lines 10-12 of Algorithm 3).

In order to compute the set of orphan messages and their phase, each process maintains a table called \( RPP \) (stands for Received Per Phase). It contains as many entries \( RPP[j] \) as incoming channels for this process. For each entry, the process keeps the date of the last received message \( RPP[j][Maxdate] \) and the phase and date of all received messages on this channel \( RPP[j][date].phase \) (line 13-14 of Algorithm 1). All the messages in \( RPP[j] \) that have a date 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 by the rolled back processes to detect the orphan messages it sends during recovery (lines 14-15 of Algorithm 2).

### D. Orchestrating Recovery

To ensure that, after a failure, messages are replayed in the causal order, a process called recovery process is launched. Algorithm 4 describes this process. 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).

The notification to a process restarting from a checkpoint or to a non-rolled back process is needed only for the first message because a process can send it when all orphan
Algorithm 3 Non-Rolled Back Processes Algorithm

Local Variables:
1: \( P_i \), \( Date_i \), \( Phase_i \), \( Cluster_i \), \( RPP_i \), \( Logs_i \)
2: \( RollbackDate \) \( \leftarrow [1, \ldots, \bot] \) \{ RollbackDate\( [j] \) 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: \( OrphPhases_i \) \( \leftarrow \emptyset \) \{ Set of phases of each orphan message \}

6: Upon failure of process \( P_j \) \( \notin Cluster_i \)
7: wait until receiving \( \langle Rollback, Date_{\epsilon_k} \rangle \) from all \( P_k \) \( \in Cluster_i \)
8: for all \( P_k \) \( \in Cluster_i \) do
9: \( Send(lastDate, RPP_i[j], MaxDate) \) to \( P_k \)
10: for all \( (P_k, Date, Phase, msg) \in Log \) such that \( Date > Date_{\epsilon_k} \) 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 \( date > Date_{\epsilon_b} \) 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 \( \langle NotifySendMsg, Phase \rangle \) from the recovery process
19: Use Algorithm 1 functions

20: Upon receiving \( \langle Rollback, Date_{\epsilon_b} \rangle \) from \( P_j \)
21: \( RollbackDate[P_j] \) \( \leftarrow Date_{\epsilon_b} \)

22: Upon receiving \( \langle NotifySendLogg, Phase_{\epsilon_{noti}} \rangle \) from the recovery process
23: for all \( (P, Date, msg) \in ResentLogs_i \) such that \( Phase \leq Phase_{\epsilon_{noti}} \) do
24: \( Send(msg, Date, Cluster_i) \) to \( P \)

messages its state depends on are received. Thus, when it can send the first message, it means that its state does not depend on orphan messages anymore: no more notifications are needed. In Figure 4, if the both \( Cluster_2 \) and \( Cluster_4 \) fail, the process \( pr \) is blocked wait for the notification to send \( m_r \). After the orphan \( m_r \) is notified, the process \( pr \) does not depend on any other orphan message, and so does not need any other notification 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 phases of the logged messages they should send (lines 10-15 of Algorithm 3) and the number of orphan messages they should receive in each phase (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 cannot send any new message until there are no more orphan messages in a lower phase than its current phase, 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 in phase \( \rho \) are received (lines 15-16 of Algorithm 4), the \( NotifySendLogg \) (notification for logged messages) and \( NotifySendMsg \) (notification for non-logged messages) notifications for the next phases with no orphans are sent (lines 18-21 and lines 22-24 of Algorithm 4). These notifications contain the phase number of the message or process to be notified.

If the recovery process would fail 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, the last checkpoint is the only one 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 by an acknowledgment, containing its current date, 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 the date contained in the acknowledgment.
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 our protocol 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 \( \sigma^k_i \), then \( Ph(e) = Ph(\sigma^k_i) \).

A. Protocol Characteristics

In this section, we provide some lemmas related to phases.

**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 \( P_i \): \( e \) and \( e' \) can be send or recv events. If event \( e^k_i = send(m) \), \( Ph(\sigma^{k-1}_i) = Ph(\sigma^k_i) \). If event \( e^k_i = recv(m) \), \( Ph(\sigma^{k-1}_i) \leq Ph(\sigma^k_i) \) (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 \( i \), i.e., we have \( Ph(e_0) \leq Ph(e_i) \). 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 and thus does not contain any orphan message, orphan messages are then 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), according to line 11-12, then \( Ph(e_0) < Ph(e_1) \). Since \( Ph(e_1) \leq Ph(e_i) \), \( Ph(e_0) < Ph(e_i) \).

**Lemma 4:** In any correct execution, send events have the same phase.

**Proof:** We assume \( send(m) \) is event \( e^k_i \) 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 \( \mathcal{S} \) is the set of correct executions: for each \( p \in P \) and \( VS \in \mathcal{S} \), \( S[p] \) contains the same sub-sequence of send events. It implies that for each \( p \in P \) and \( VS \in \mathcal{S} \), \( S[p] \) contains the same set of recv events. Since events on a process are ordered by the happened-before relation, \( S[p_i] \) contains the same set of recv events \( \forall S \in \mathcal{S} \). Thus \( Ph(send(m)) \) is the same in any correct execution.

B. Consistency of the Sequence of Events

Let \( \Sigma_r \) 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 \( e^k_i = send(m) \) on process \( p_i \) can happen in state \( \Sigma_r \). \( \Sigma_r \) consists of correct executions for each \( p \in P \) and \( VS \in \mathcal{S} \), \( S[p] \) contains the same sub-sequence of send events. It implies that for each \( p \in P \) and \( VS \in \mathcal{S} \), \( S[p] \) contains the same set of recv events. Since events on a process are ordered by the happened-before relation, \( S[p_i] \) contains the same set of recv events \( \forall S \in \mathcal{S} \). Thus \( Ph(send(m)) \) is the same in any correct execution.
$Ph(send(m'')) < Ph(send(m'))$. 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 $m$ orphans (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 resent.

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

**Proof:** Let $\mathcal{O}$ be the set of orphan messages in state $\Sigma$. Let $\min\_phase$ be the smallest $Ph(send(m)), \forall m \in \mathcal{O}$ . According to lines 18-21 and 22-24 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 $m \in \mathcal{O}$ with $Ph(send(m)) = \min\_phase$ can eventually be replayed. Then the same reasoning applies to the new $\min\_phase$, until $\mathcal{O} = \emptyset$.

V. Evaluation

We implemented HydEE in the MPICH2 library. 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 sub-system 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. This is why 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 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.

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, with and without message logging, 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, and takes as input a graph defining the amount of data sent on 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. It tends to show that hybrid protocols based on clustering might becomes more interesting when the applications increase in scale.

<table>
<thead>
<tr>
<th>Nb Clusters</th>
<th>Avg Ratio of Process to Roll Back (Single Failure Case)</th>
<th>Log/Total Amount of data (in GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAS BT</td>
<td>5</td>
<td>21.78%</td>
</tr>
<tr>
<td>NAS CG</td>
<td>16</td>
<td>6.25%</td>
</tr>
<tr>
<td>NAS FT</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>NAS LU</td>
<td>8</td>
<td>12.5%</td>
</tr>
<tr>
<td>NAS MG</td>
<td>4</td>
<td>25%</td>
</tr>
<tr>
<td>NAS SP</td>
<td>6</td>
<td>18.56%</td>
</tr>
</tbody>
</table>

C. Communication Performance

Figure 5 compares MPICH2 native communications performance over Myrinet 10G, to the performance provided by HydEE with and without message 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 plateaus in the native performance of MPICH2 over MX. For instance, in our experiment, the native latency of MPICH2 is around 3.3 \( \mu s \) for messages size 1 to 32 bytes and then jump to 4 \( \mu s \). Because of the message size increase due to the additional data sent, HydEE reaches these plateau earlier. Second, the performance with and without sender-based message 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 topic 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.

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 may slowdown the application during failure free execution [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. We proposed an uncoordinated checkpointing protocol without domino effect for send-deterministic applications [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, the 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 of one cluster affects only this clusters. In all these protocols, the determinants of all messages have to be logged [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.

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. The communication patterns of most MPI HPC applications allow to cluster application processes with a good trade-off between clusters size and amount of inter-cluster communications. 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. Finally, we plan to study how our protocol could be integrated with topology-aware multi-level checkpointing techniques [15].

REFERENCES


