Reasons for a Pessimistic or Optimistic Message Logging Protocol in MPI Uncoordinated Failure Recovery

Aurelien Bouteiller [†1], Thomas Ropars [◇2], George Bosilca [†3], Christine Morin [*4], Jack Dongarra [†5]

[†] *Innovative Computing Laboratory, University of Tennessee*
1122 Volunteer Boulevard, Knoxville TN, 37996, USA
[1] bouteill, [3] bosilca, [5] dongarra@eecs.utk.edu

[◇] *Université de Rennes 1, IRISA*
INRIA, Centre Rennes - Bretagne Atlantique
Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
[2] tropars@irisa.fr

[*] *INRIA, Centre Rennes - Bretagne Atlantique*
Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
[4] christine.morin@inria.fr

***Abstract*—With the growing scale of high performance computing platforms, fault tolerance has become a major issue. Among the various approaches for providing fault tolerance to MPI applications, message logging has been proved to tolerate higher failure rate. However, this advantage comes at the expense of a higher overhead on communications, due to latency intrusive logging of events to a stable storage. Previous work proposed and evaluated several protocols relaxing the synchronicity of event logging to moderate this overhead. Recently, the model of message logging has been refined to better match the reality of high performance network cards, where message receptions are decomposed in multiple interdependent events. According to this new model, deterministic and non-deterministic events are clearly discriminated, reducing the overhead induced by message logging. In this paper we compare, experimentally, a pessimistic and an optimistic message logging protocol, using this new model and implemented in the Open MPI library. Although pessimistic and optimistic message logging are, respectively, the most and less synchronous message logging paradigms, experiments show that most of the time their performance is comparable.**

I. Introduction and Motivation

Recently, High Performance Computing (HPC) platforms have reached such a critical scale that failures actually impact the usability of the systems. Consequently many HPC centers have been shifting their attention toward capacity, providing a massive amount of computation time to the community, rather than capability. Because of the difficulty of coping with failures and scale, only a small number of highly tuned applications are able to benefit from the entire capability of the machines. As most HPC applications are using the Message Passing Interface (MPI) [1] to manage data transfers, introducing failure recovery features inside the MPI library automatically benefits a large range of applications. One of the most popular automatic fault tolerant techniques, coordinating checkpoint, builds a consistent recovery set [2], [3]. Message logging is an alternative approach designed to avoid coordination, in order to recover faster from failures at the expense of a higher overhead on communications. From previous results [2], message logging is expected to have a better typical application makespan than coordinated checkpoint when the Mean Time Between Failure (MTBF) is shorter than 9 hours. Moreover, while coordinated checkpoint stalls when the MTBF is shorter than 3 hours, message logging can still progress.

Over the years, different versions of message logging have been proposed to address the issue of high latency associated with synchronous logging of events to a stable storage [4]. However, the model of message logging was recently refined to match the reality of high performance network interface cards, where message receptions are decomposed in multiple interdependent events [5]. From a finer decomposition of events impacting the life-cycle of a MPI message, the need for intermediate message copies impacting bandwidth on high performance networks is lifted; deterministic and non-deterministic events are clearly discriminated, allowing a reduction of the overall number of messages requiring latency disturbing management.

In this paper, we present two implementations of message logging, pessimistic and optimistic message logging, respectively being the most and the less synchronous possible paradigms, based on a generic failure recovery framework implementing the aforementioned improvements in a leadership MPI implementation (Open MPI [6]). Then we perform a comprehensive experimental comparison of those two approaches using micro-benchmarks and exploring their behavior on a wide range of scientific kernels. Results demonstrate how improvements targeted at adapting message logging to high performance networks have dramatically altered the

978-1-4244-5012-1/09/$25.00 ©2009 IEEE

knowledge acquired in previous work about the impact of synchronicity on event logging performance.

The rest of this paper is organized as follows: in the next section we recall the classical message logging model. Then in the third section we depict the modifications introduced to better fit HPC networks and some details about the implementation in a shared framework in the fourth section. The fifth section presents a comparison of experimental results, and is followed by the related works and conclusion.

II. Description of Message Logging

Message logging is defined in the more general model of message passing distributed systems. Communications between processes are considered explicit: processes explicitly request sending and receiving messages; and a message is considered as delivered only when the receive operation associated with the data movement is complete. Additionally, from the perspective of the application each communication channel is FIFO, but there is no particular order on messages traveling along different channels. The execution model is pseudo-synchronous; there is no global shared clock among processes but there is some (potentially unknown) maximum propagation delay of messages in the network. An intuitive interpretation is to say the system is asynchronous and there is an *eventually reliable* failure detector. Failures can affect both the processes and the network. Usually, network failures are managed by some CRC mechanism and message reemission provided by the hardware or low level software stack and do not need to be considered in the model. Therefore, the considered failure model is definitive crash failures, where a failed process completely stops sending any subsequent message.

A. Events

Each computational or communication step of a process is an event. An execution is an alternate sequence of events and process states, with the effect of an event on the preceding state leading the process to the new state. As the system is basically asynchronous, there is no direct time relationship between events occurring on different processes. However, Lamport defines a causal partial ordering between events with the *happened before* relationship [7]. It is noted $e \prec f$ when event f is causally influenced by e.

These events can be classified into two categories: deterministic and non-deterministic. An event is deterministic when from the current state there is only one possible outcome state for this event. On the contrary, if an event can result in several different states, then it is non-deterministic. Examples of deterministic events are internal computations and message emissions, which follow the code-flow. Examples of non-deterministic events are message receptions, which depend on time constraints on message deliveries.

B. Checkpoints and Inconsistent States

Checkpoints (i.e., complete images of the process memory space) are used to recover from failures. The recovery line is

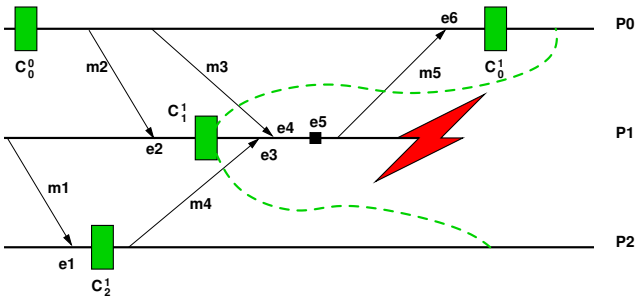


Fig. 1. Example execution of a distributed system with checkpoints and inconsistent recovery line.

the configuration of the application after some processes have been reloaded from checkpoints. Unfortunately, checkpointing a distributed application is not as simple as storing each single process image without any coordination, as illustrated by the example execution of figure 1. When process P_1 fails, it rolls back to checkpoint C_1^1. Messages from the past crossing the recovery line (m_3, m_4) are *in transit* messages; the restarted process will request their reception while the source process never sends them again, thus it is needed to save these messages. Messages from the future crossing the recovery line (m_5) are orphan; following the Lamport relationship current state of P_0 depends on reception of m_5 and by transitivity on any event that occurred on P_1 since C_1^1 (e_3, e_4, e_5). Since the channels between P_0 and P_1 and between P_2 and P_1 are asynchronous, the reception of m_3 and m_4 could occur in a different order during re-execution, leading during the recovery to a state of $P1$ that diverges from the initial execution. As the current state of P_0 depends on states P_1 can no longer reach, the overall state of the parallel application after the recovery is inconsistent. Checkpoints leading to an inconsistent state are useless and must be discarded. In the worst case all checkpoints are useless and the computation may have to be restarted from the beginning.

C. Event Logging

In event logging, processes are considered as *Piecewise deterministic*: only sparse non-deterministic events occur separating large parts of deterministic computation. Considering that non-deterministic event outcomes, called determinants, are committed during the initial execution into some *safe* repository, a recovering process is able to replay exactly the same order for all non-deterministic events, and therefore, it is able to reach exactly the same state as prior to the failure. Furthermore, message logging considers the network as the only source of non-determinism and only logs the relative ordering of messages from different senders (e_3, e_4 in figure 1). The sufficient condition to define a consistent global state, from where a recovery can be successful, is that a process must never depend on an unlogged non-deterministic event from another process.

D. Synchronicity of Event Logging

Pessimistic message logging is the most synchronous event logging technique. It ensures the always no-orphan condition: all the previous non-deterministic events of a process must be logged before a process is allowed to impact the rest of the system. Therefore any process has to ensure that every event is safely logged before any MPI send can proceed. Since no orphan process can be created, only the failed processes have to restart after a failure. In order to improve latency, the no-orphan condition can be relaxed. Causal message logging piggybacks unlogged events on outgoing messages. Then any process always depends on events either logged or known locally. Optimistic message logging pushes one step further; non-deterministic events are buffered in the process memory and logged asynchronously. While message sending is never delayed, the consequence is that a message sent by a process may depend on an unlogged event and may become orphaned. Thus a recovery protocol is needed to detect orphan messages and to recover the application in a consistent global state after a failure. To be able to detect orphan messages, dependencies between non-deterministic events need to be tracked during the entire execution; dependency information must be piggybacked on application messages.

E. Sender-Based Logging

Event logging only saves events in the remote repository, without storing the message payload. However, when a process is recovering, it needs to replay any reception that happened between the last checkpoint and the failure. Therefore, all the in transit message payload needs to be saved (m_3, m_4 in figure1). During normal execution, every outgoing message is saved in the sender's volatile memory: a mechanism called sender-based message logging. This allows the surviving processes to serve past messages to recovering processes on demand, without rolling back. Unlike events, sender-based data do not require stable or synchronous storage. Should a process holding useful sender-based data crash, the recovery procedure of this process replays every outgoing send and thus rebuilds the missing messages.

III. Recent Amendments to Message Logging

Though the classic model of message logging has been extensively evaluated in the past, it has been strongly revamped recently to adapt to the context of MPI on high speed interconnects. The modifications root deeply into the model itself and drastically change the essence and the repartition of the overhead endured by message logging. In this section we describe the nature of the changes at the source of our motivation for a new evaluation of the impact of synchronicity on event logging performance.

A. Adapted Message Logging Model for MPI Communications

Though the previous model has been used in many implementations of message logging in the past, it is unable to capture the complexity of MPI communications. This was left unaddressed as long as the performance gap between network and memory bandwidth was hiding the ensuing overhead. But as the performance of network interface cards progressed it became clear that extra memory copies on the critical path of messages were the source of significant performance penalties. Discrepancies between the model and the reality of MPI communication basically lie in the existence of non-blocking communications. Those are intended to maximize opportunities for communication overlap by computation by allowing for the application to post its intention to communicate, compute while the communication actually takes place, and to wait for completion of the communications later. The rest of this section details the improved model used to better describe non-deterministic events with concurrent non-blocking messages.

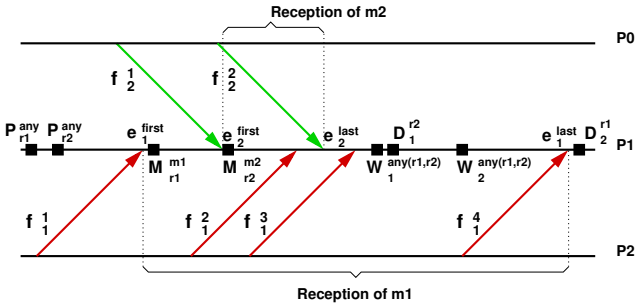


Fig. 2. Consequence of message fragmentation on event ordering of non blocking MPI receives

1) Fragments: Every message is divided into a number of network fragments when it is transfered over the network, the number depending on its length. Though MPI enforces a FIFO semantic for messages from a particular sender, at the lowest network level there is no particular order between fragments. Consequently, as depicted in the example of figure 2, when receiving two different messages m_1 and m_2, the first fragment of m_1 coming first does not imply that the last fragment of m_1 arrives before the last fragment of m_2. Therefore, unlike in the classic model, with MPI communications the reception order of a message cannot be fully described by a single event denoting *message reception*, but rather depends on the relative ordering of the multiple fragments composing the messages. Although there is a very large number of such network non-deterministic events, only the order of events denoting the first and last fragments of messages are actually meaningful to the application, as described in the next paragraphs.

2) Matching: In order to receive a message, an MPI application needs to post a reception request, using the `MPI_Irecv` or `MPI_Recv` functions. Each request contains a buffer, a source, a tag and its relative ordering to other requests, depending on the date it has been posted. When the first fragment of a message is delivered by the network, requests are considered in order by the matching logic; the first request with a matching source and tag is associated with the incoming message fragments. All upcoming fragments of this message are delivered directly into the request's reception buffer. If no request matches, the message is *unexpected*; it

is copied into an internal buffer until it matches an upcoming posted request.

A *matching determinant* is the event denoting the association between the first fragment of a message and a particular request. In the example of figure 2, $M_{r_1}^{m_1}$ is the matching determinant between the request created by the any-source non-blocking receive $P_{r_1}^{any}$ and the first fragment reception event e_1^{first}. Though the relative order of the fragments from the network is always non-deterministic, the FIFO by channel MPI semantic allows for most of the matching determinants to be deterministic. The only non-deterministic ones are promiscuous receptions, *i.e.*, when a request can match a message coming from any-source. Those promiscuous matching determinants are the only events that need to be logged in order to replay a correct matching during recovery.

3) Waiting for completion of requests: When using non-blocking communications, several requests can concurrently progress while the application is computing. When computation cannot process further without accessing buffers involved in an ongoing communication, the application waits for the completion of the corresponding requests. All the functions allowing the application to check the status of a request (like `MPI_Wait`) are represented by a *completion test* event. A *delivery determinant* is the event denoting the association between a particular completion test event and a message last fragment event. As an example, in figure 2, $D_1^{r_2}$ is the delivery determinant associated to the last fragment reception event e_2^{last} and the completion test $W_1^{any(r_1,r_2)}$. A special *bottom* event denotes that no last fragment event occurred since the last test for completion event.

Again, the most common delivery determinants are always deterministic, namely the `MPI_Recv`, `MPI_Send`, `MPI_Wait` and `MPI_Waitall` functions. However, for `MPI_Waitany`, the outcome of the MPI call depends on the ordering between last fragment events of messages matched with the waited requests. `MPI_Waitsome`, `MPI_Test`, `MPI_Testany`, `MPI_Testsome` and `MPI_Iprobe` add to the previous source of non-determinism a dependency between the arrival date of the last fragments and the date of the completion test. Logging all the delivery determinant events appearing in a function where only a subset of the requests is allowed to complete is sufficient to ensure a deterministic replay of all non-deterministic deliveries.

4) Benefits from the improved model: One of the most important optimizations for a high throughput communication library is *zero copy*: the ability to send and receive directly into the application's user-space buffer without intermediary memory copies. To enable it, the matching must be resolved upon arrival of the very first fragment. When it is delayed until the completion of the message, as it is necessary when using the legacy model of atomic *message reception event*, the actual result is that the message cannot be delivered directly into the application buffer. The MPI library has not yet associated a request with the message; every message pays the same penalty as if it were unexpected.

The only software layer where the MPI matching can be delayed is the very low level interface with the network. Implementing message logging at this level has two severe limitations. First the message logging mechanism cannot easily take advantage of the optimized network drivers and second, at this level it is impossible to make a distinction between deterministic and non-deterministic delivery determinants. By interposing the event logging mechanism higher in the MPI library architecture, it is only necessary to log the communication events at the library level, and one can completely ignore the expensive events generated by the lower network layer, overall reducing by a large amount the number of events to log.

B. Active Optimistic Message Logging

A new optimistic message logging solution, called active optimistic message logging [8], has been recently proposed to limit the drawbacks of existing optimistic message logging protocols.

Optimistic message logging has two main drawbacks. First, it is less efficient than pessimistic message logging on recovery because orphan processes may be created. In the event of a failure, a recovery protocol must be executed to detect orphan processes and these orphan processes must be rolled-back in addition to the failed processes. Second, to track dependencies between processes during failure free execution, dependency information must be piggybacked on application messages, adding overhead on communications [9].

In the standard model of optimistic message logging, determinants are buffered is the process memory and logged asynchronously. O2P is an active optimistic message logging protocol, *i.e.*, it logs non-deterministic determinants on stable storage as soon as possible to reduce the probability that a message depends on an unlogged determinant when it is sent. Thus it reduces the risk of orphan message creation in case of failure.

To reduce the amount of data piggybacked on application messages, it has been proved that to be able to detect orphan messages only dependencies to unlogged non-deterministic determinants have to be tracked [10]. Since active optimistic message logging maximizes the probability that previous non-deterministic determinants are logged when a message is sent, it reduces the amount of data that needs to be piggybacked on application messages.

IV. Implementation Details

This section details the implementation of the two protocols in Open MPI. First, we present the `Vprotocol` framework, shared by the two implementations, that provides the basic blocks of message logging. Then we focus on some details of the O2P implementation, particularly the modifications to the event logger to manage the optimist protocol.

A. The Shared Framework

The `Vprotocol` framework enables the implementation of message logging protocols in the Open MPI library. It is based on the refined model presented in the previous section.

The two protocols compared in this paper are implemented in this framework, allowing for a fair and equitable comparison. In this section, we present the common parts of the two implementations, through the description of the `Vprotocol` framework main functionalities, *i.e.*, sender-based message logging, remote event storage, any-source reception event logging, and non-deterministic delivery event logging.

1) Sender-Based Logging: When a message is sent by a process, its payload is saved locally. Thus in the event of failure, the message can be replayed by the process according to data available in its memory. If the process fails, these data are lost but will be regenerated during recovery. Sender-based message logging avoids the costly copy of the message payload on a stable storage.

The sender-based logging is integrated into the data-type engine of Open MPI. The data-type engine is in charge of packing (possibly non-contiguous) data into a flat format suitable for the receiver's architecture. Each time a fragment of the message is packed, the resulting data is copied in a *mmaped* memory segment. Because the sender-based copy progresses at the same speed as the network, it benefits from cache reuse and releases the send buffer at the same time. Data is then asynchronously written from memory to disk in the background to decrease the memory footprint.

2) Event Logger: Non-deterministic events are sent to the event loggers processes (EL). An EL is a special process added to the application outside of the MPI_COMM_WORLD; several might be used simultaneously to improve scalability. Events are transmitted using non blocking MPI communications over an inter-communicator between the application process and the event logger. A transactional acknowledgement protocol is used to make application processes aware of logged events.

3) Any-Source Receptions: When an any-source receive request is completed, a new event is logged containing the request identifier and the matched source. Because channels are FIFO, enforcing the source during recovery is enough to replay the original matching order.

4) Non-Deterministic Deliveries: Every non-deterministic completion test is assigned a unique clock. A delivery event containing the list of requests delivered by the operation is created for this clock. During replay, when the completion test's clock is equal to the clock of the first event, the corresponding requests are completed by waiting for each of them.

Should the outcome of the completion test be that no request completed, to avoid the creation of a large number of events for consecutive unsuccessful completion test, we use lazy logging; only one event is created for all the consecutive operations. If a completion test succeeds, any pending lazy event is discarded. During recovery, any completion test whose clock is lower than the first event in the log has to return that no request completed.

B. O2P Implementation Details

1) Dependency Tracking: To track dependencies between application processes, O2P uses a dependency vector. A dependency vector is an n entry vector, n being the number of processes in the application. Entry j of process p_i dependency vector is the last unlogged non-deterministic event of process p_j that the current state of p_i depends on. If entry j is empty, it means that p_i doesn't depend on any unlogged non-deterministic event from the process p_j. When a process sends a message, it piggybacks its dependency vector on the message. The process receiving that message updates its own dependency vector with the piggybacked vector.

When a non deterministic event occurs at process p_i, it sends the event to the EL and saves it in entry i of its dependency vector. This entry is emptied by the process when it receives the acknowledgement from the EL. To limit the piggybacked data size, dependency vectors are implemented as described in [11]. Only non-empty entries that have changed since the last message sent to the same process are piggybacked on the message.

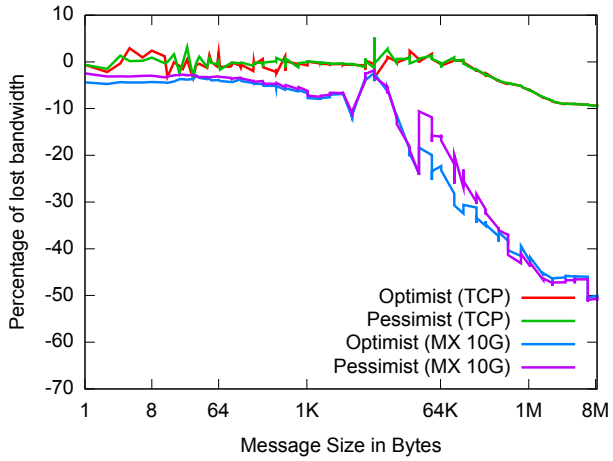
2) Event Logger: In order to make a process aware of the events saved by other processes, the EL maintains a n entry vector that we call the stable vector. Entry k of the stable vector is the last event of process p_k the EL has received. The stable vector is included in the acknowledgements sent by the EL. When a process delivers an acknowledgement from the EL, it updates its dependency vector according to the stable vector received. This mechanism contributes to reduce the size of the piggybacked data.

3) Piggyback mechanisms: Piggyback mechanisms have a significant impact on O2P failure free performance. Due to active optimistic message logging, most of the time there is no data to piggyback on application messages. That's why we have implemented a solution that optimizes this case. Piggybacked data are sent in a separate message. An additional flag is included in the application message header to make the destination process aware of the presence of piggybacked data. Thus an additional message is sent only if there is data to piggyback.

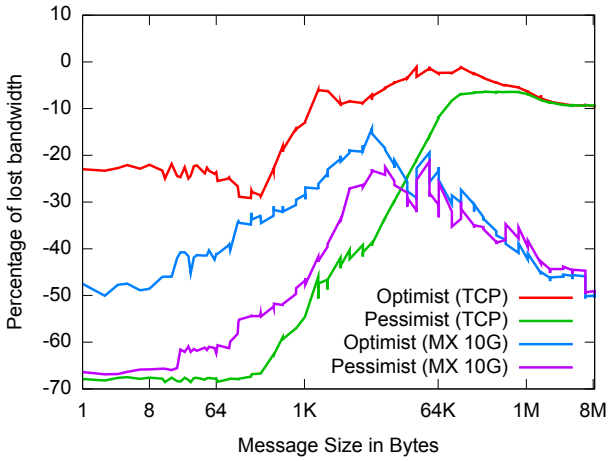
V. Experimental Comparison

In this section, we compare the performance obtained by the optimistic and pessimistic protocols taking into account the impact of the new message logging model. NetPIPE [12] is used to benchmark the ping-pong bandwidth and latency. To investigate application performance we use the NAS Parallel Benchmark suite [13], a set of kernels and applications provided by the NASA NAS research center that covers a large panel of communication schemes and application patterns. For our experiments we used 6 of the NAS benchmarks: BT, CG, FT, LU, MG and SP. The results presented are mean values over 5 executions of each test.

The experiments were run on a 138 node cluster belonging to the Grid'5000 testbed [14], composed of 27 Dell PowerEdge 1950 servers equipped with an Intel Xeon 5148 LV processor running at 2.33 Ghz, 8 GB of memory and a 300 GB SATA



(a) Common case without non-deterministic events.



(b) With forced non-deterministic events.

Fig. 3. Ping-pong performance comparison of pessimistic and optimistic protocols.

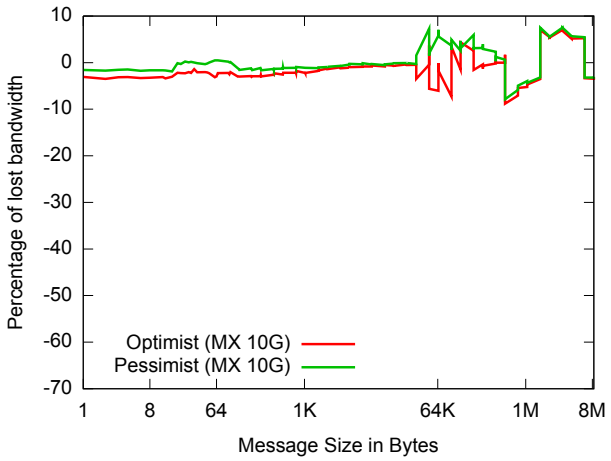


Fig. 4. Myrinet 10G ping-pong performance of pessimistic and optimistic protocols without sender-based payload logging.

hard drive; 63 Dell PowerEdge 1950 servers equipped with an Intel Xeon 5148 LV processor running at 2.33 Ghz, 4 GB of memory and a 160 GB SATA hard drive; and 38 HP ProLiant DL145G2 servers equipped with an AMD Opteron 246 processor running at 2.0 Ghz, 2 GB of memory and a 80 GB SATA hard drive. All the nodes had a Gigabit Ethernet interface and were connected by a single Cisco 6509 switch. Ninety of the Dell PowerEdge nodes were additionally connected to a single Myrinet 10G switch. Linux 2.6.18 was the operating system with the mx-1.2.0j Myrinet driver.

A. Ping-pong Performance

For this set of experiments, NetPIPE is deployed on two Dell PowerEdge 1950 servers while a third one hosts the Event Logger. The results of figure 3(a) show a regular Gigabit-Ethernet ping-pong for the two protocols. With the default options, there is no non-deterministic event in this benchmark. Therefore, thanks to the optimizations introduced by the refined model, there is no event to log and the latency overhead is unnoticeable. As a consequence, both protocols exhibit very similar behavior.

a) Impact of non-deterministic events: To investigate the impact of event logging, we force any source receptions in the NetPIPE benchmark. According to the pattern of communication in this benchmark, a non-deterministic event is created by each reception and is immediately followed by a send, forcing the pessimistic protocol to log an event synchronously before allowing the send to proceed. Figure 3(b) illustrate that the consequence is a threefold increase in latency. The overhead induced by the optimistic protocol is much smaller: while the event is still sent to the Event Logger immediately, the next send does not need to be delayed until the reception of the acknowledgement. The impact of piggybacked data management is very small, as the application has only two processes, the maximum number of events to piggyback is at most one.

b) High performance networks: Focusing on the Myrinet 10G network results from figure 3(a), the very low latency of both protocols illustrates that without non-deterministic events, the cost of event logging is well contained on high performance networks. As seen in figure 3(b), the relative overhead of managing non-deterministic events is not modified; the pessimistic protocol still endures a threefold increase in latency while the optimistic one sees a milder degradation. However, the performance penalty associated with sender-based payload logging, a shared characteristic of all message logging protocols, increases as the network becomes faster. The Myrinet network is fast enough that even being asynchronous, the extra memory copy generated by the sender-based payload logging drains more memory bandwidth than available. Experiments where the sender-based mechanism is disabled, depicted in figure 4, further support that explanation, with no bandwidth degradation compared to non fault tolerant MPI.

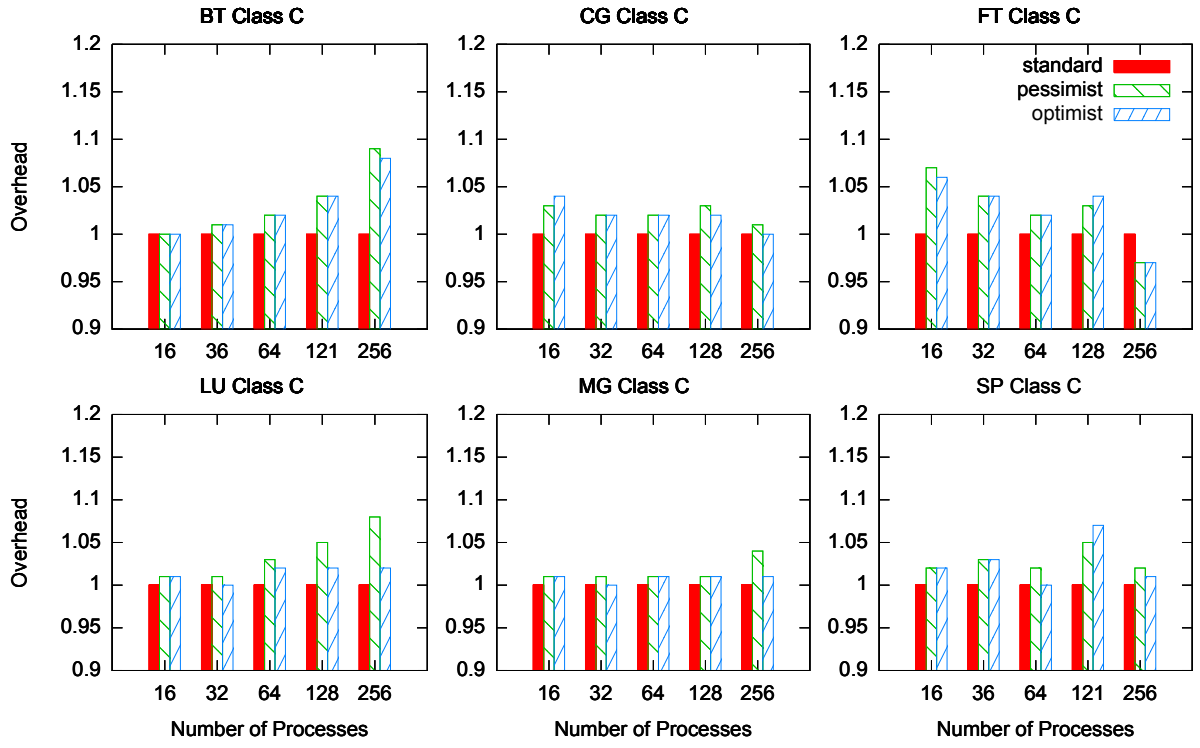


Fig. 5. Scalability comparison of pessimistic and optimistic message logging protocols on the NAS Benchmarks on Gigabit Ethernet

B. Scalability

In order to evaluate the comparative scalability of the two protocols we plot the normalized execution time of the NAS kernels according to a growing number of processors (figure 5). While in previous experiments (figures 3) we specifically outlined the differences caused by non-deterministic events, in this phase of the comparison we focus on widely used application kernels. Among the NAS kernels, only two generates non deterministic events: MG and LU. As a consequence, the executions of the two protocols are very similar and exhibit the same scalability. Overall, the overhead induced by the sender-based payload copy mechanism stays under 10% on these benchmarks.

The only benchmark showing a different scalability pattern is LU. The number of non-deterministic events grows with the size of the application, making the optimistic protocol 6% more efficient than the pessimistic one for 256 processes.

C. Isolating Event Logging Overhead

Figure 6 presents the performance of all the NAS kernels for 64 processes on the Myrinet network. Every kernel is evaluated with or without the sender-based mechanism being active. While it is a required component for a successful recovery, deactivating the sender-based overhead reveals the performance differences imputable to the event logging protocols. As expected, the performance of event logging exhibits almost no differences between the protocols on the benchmarks where there is no non-deterministic events. Even on those with non-deterministic events, the performance varies

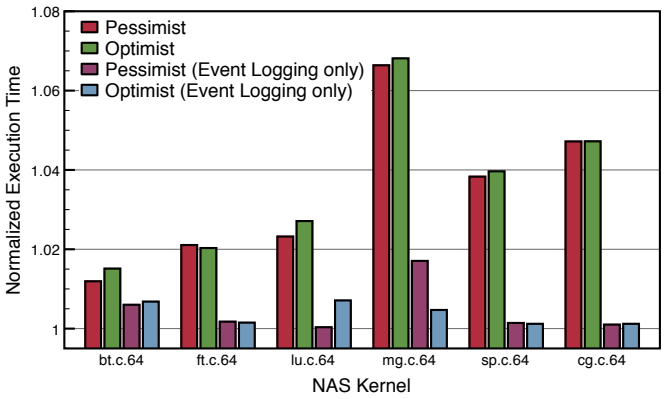


Fig. 6. Normalized performance of the NAS kernels on the Myrinet 10G network (Open MPI=1).

only by less than 2%, which is close to the error margin of measurements. On this faster network, the sender-based overhead clearly dominates the performance and flattens any performance difference coming from the synchronicity of the event logging.

D. Event Logging Overhead Breakdown

To evaluate the cost of event logging in the protocols, we used a small ping-pong test with 2 processes. The *any_source* flag was used in order to generate a non-deterministic event for every message reception. Results are presented in Table I. First, when a non deterministic event is created, it has to

	Cost of Event Sending	Cost of EL Acknowledgment Management
Pessimist	6.3 μs	97.1 μs
Optimist	6.7 μs	1.1 μs

TABLE I
COST OF A SINGLE MESSAGE GENERATING AN UNIQUE NON-DETERMINISTIC EVENT.

	Cost of Event Sending	Cost of EL Acknowledgment Management
Pessimist	17 ms	150 ms
Optimist	12 ms	42 ms

TABLE II
CUMULATIVE COST OF NON-DETERMINISTIC EVENTS MANAGEMENT IN LU.C.64.

be sent to the event logger. The cost of event sending is as expected the same with optimistic and pessimistic message logging.

The pessimistic message logging protocol then has to wait for the acknowledgement from the event logger before sending the next message whereas the optimistic protocol doesn't stop the execution of the application process to wait for the acknowledgement. The only overhead induced by the acknowledgments management in the optimistic message logging protocol is related to the update of the dependency vector on acknowledgement delivery.

Table II details the cost of event logging on class C LU NAS Parallel Benchmark for 64 processes. The cost of events sending is again almost equal between the two protocols. However, the overhead difference on acknowledgement management between the two protocols is reduced; as the size of the dependency vector depends on the total number of processes, a larger setup is prone to reduce this gap.

VI. Related Work

Though fault tolerance can be fully managed by the application [15], [16], the software engineering cost prevents a large number of applications from benefiting of the entire capacity of modern clusters. FT-MPI [17], [18] aims at helping an application to express its failure recovery policy by taking care of rebuilding internal MPI data structures (communicators, rank, etc.) and triggering user provided callbacks to restore a coherent application state when failures occur. Though this approach is very efficient to minimize the cost of failure recovery techniques, it still adds a significant level of complexity to the design and implementation of the parallel applications.

The next step toward easing application development is automatic fault tolerant MPI libraries, where failures are completely hidden from the application, thus avoiding any modification of the user's code. Consistent recovery can be achieved automatically by building a coordinated checkpoint set where no orphan message exists (with the Chandy & Lamport algorithm [19], [20], [2], CIC [21] or blocking the application until channels are empty [3], [22]). In all coordinated checkpoint techniques, the only consistent recovery set is when every process, including non failed ones, restart from a checkpoint.

Another approach that allows for faster recoveries according to [2] is to use message logging. Manetho [23], Egida [24] and MPICH-V [25] feature the main flavors of message logging (optimistic, pessimistic and causal). Because they rely on the classic message logging model, these protocols cannot distinguish between deterministic and non-deterministic events and introduce extra memory copies leading to a performance penalty on high throughput networks. Optimistic message logging protocols, such as [26], [27], [28], [10], delay the storing of determinants to the stable storage and keeps them in the process memory. As a consequence, they are more subject to creating orphan processes and to piggyback more determinants with messages. Active optimistic message logging protocol [8] copes with this drawback by aggressively saving determinants to the stable storage as soon as possible

In this paper we present state of the art pessimistic and optimistic message logging protocols. They both benefit from an improved model allowing to significantly decrease the number of memory copies on the critical path and to minimize the number of non-deterministic events to be saved. Moreover, active optimistic and pessimistic protocols have never been compared to date. Besides being the state of the art in their respective categories, these two protocols, due to their highly optimized implementations, can shed new light on the merits of event logging synchronicity. As a consequence, our experimental results suggest a radically different conclusion than previous evaluations of message logging protocols [29].

VII. Conclusion

The recent update of the message logging model, to adapt to the context of MPI on high speed interconnects, questions the results provided by previous works on message logging evaluation. In this new model, message receptions are decomposed into multiple interdependent events, allowing to clearly discriminate between deterministic and non deterministic events. Since the total number of events that need to be logged on stable storage is strongly reduced, the impact of event logging on performance is also reduced, motivating a new comparison of the impact of asynchrony on message logging.

Pessimistic and optimistic message logging are, respectively, the most and the less synchronous message logging solutions. Optimistic message logging exchanges the ability to delay logging of determinants with the need to rollback some non-failed processes during recovery. The optimistic message logging protocol, called O2P, is an active optimistic message logging protocol introducing an aggressive logging strategy to reduce the impact of optimistic message logging on both failure free execution and recovery. In this paper we have evaluated the impact of the new message logging model through the comparison of state of the art pessimistic and optimistic message logging protocols based on this new model and implemented in the open MPI library.

When some non-deterministic events need to be logged, optimistic message logging doesn't require any synchronization with the EL. As outlined by NetPIPE and the event handling overhead breakdown, this allows optimistic logging to reach a twofold better latency in that case. When considering some of the most useful application kernels, the performance degradation due to synchronous message logging is very limited. When the application actually uses non-deterministic communication patterns, a five to six percent difference can be measured between the two protocols. The trend on LU suggests that this difference could become significant at larger scale. However, the new message logging model significantly reduces the number of events that need to be saved to a reliable storage. For four of the six representative benchmarks we tested, there is even no non-deterministic events left to log. As a result, the overhead induced by message logging on failure free execution is very low and is mainly the consequence of sender-based logging. Being pessimist and optimist in that case is not important anymore since the two message logging protocols provide equivalent performance.

From a broader perspective, with the combined effect of improvement of event logging and increase of the network interface performance, sender-based payload copy is now the dominant overhead of message logging, further flattening differences between different event logging protocols. Future research efforts are needed toward defining solutions to either reduce the amount of sender-based data copied, or reduce the cost of the copy. The latest could be achieved by transferring the burden of making sender-based copies from the processor to a DMA capable component of the system, such as a GPU or a chipset.

REFERENCES

[1] The MPI Forum, "MPI: a message passing interface," in *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 1993, pp. 878–883.

[2] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, "Improved message logging versus improved coordinated checkpointing for fault tolerant MPI," in *IEEE International Conference on Cluster Computing (Cluster 2004)*. IEEE CS Press, 2004.

[3] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," in *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

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

[5] A. Bouteiller, G. Bosilca, and J. Dongarra, "Redesigning the message logging model for high performance," in *International Supercomputer Conference (ISC 2008)*, Dresden, Germany, June 2008.

[6] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[8] T. Ropars and C. Morin, "O2P: An Extremely Optimistic Message Logging Protocol," INRIA Research Report 6357, November 2007.

[9] M. Schulz, G. Bronevetsky, and B. R. Supinski, "On the Performance of Transparent MPI Piggyback Messages," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 194–201.

[10] O. P. Damani, Y.-M. Wang, and V. K. Garg, "Distributed Recovery with K-optimistic Logging," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 1193–1218, 2003.

[11] M. Singhal and A. Kshemkalyani, "An Efficient Implementation of Vector Clocks," *Information Processing Letters*, vol. 43, no. 1, pp. 47–52, 1992.

[12] 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.

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

[14] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[15] A. Roy-Chowdhury and P. Banerjee, "Algorithm-based fault location and recovery for matrix computations on multiprocessor systems," *IEEE Trans. Comput.*, vol. 45, no. 11, pp. 1239–1247, 1996.

[16] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault tolerant high performance computing by a coding approach," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM Press, 2005, pp. 213–223.

[17] G. Fagg and J. Dongarra, "FT-MPI : Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *7th Euro PVM/MPI User's Group Meeting2000*, vol. 1908 / 2000. Balatonfred, Hungary: Springer-Verlag Heidelberg, september 2000.

[18] G. E. Fagg, A. Bukovsky, and J. J. Dongarra, "HARNESS and fault tolerant MPI," *Parallel Computing*, vol. 27, no. 11, pp. 1479–1495, October 2001.

[19] K. M. Chandy and L. Lamport, "Distributed snapshots : Determining global states of distributed systems," in *Transactions on Computer Systems*, vol. 3(1). ACM, February 1985, pp. 63–75.

[20] G. Stellner, "CoCheck: Checkpointing and process migration for MPI," in *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*. Honolulu, Hawaii: IEEE CS Press, April 1996.

[21] J.-M. Hlary, A. Mostefaoui, and M. Raynal, "Communication-induced determination of consistent snapshots," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 9, pp. 865–877, 1999.

[22] J. F. Ruscio, M. A. Heffner, and S. Varadarajan, "Dejavu: transparent user-level checkpointing, migration and recovery for distributed systems," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006, p. 158.

[23] Elnozahy, Elmootazbellah, and Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output," *IEEE Transactions on Computing*, vol. 41, no. 5, May 1992.

[24] S. Rao, L. Alvisi, and H. M. Vin, "Egida: An extensible toolkit for low-overhead fault-tolerance," in *29th Symposium on Fault-Tolerant Computing (FTCS'99)*. IEEE CS Press, 1999, pp. 48–55.

[25] A. Bouteiler, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V project: a multiprotocol automatic fault tolerant MPI," vol. 20. SAGE Publications, Summer 2006, pp. 319–333.

[26] R. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Transactions on Computing Systems*, vol. 3, no. 3, pp. 204–226, 1985.

[27] A. P. Sistla and J. L. Welch, "Efficient Distributed Recovery Using Message Logging," in *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 1989, pp. 223–238.

[28] S. W. Smith, D. B. Johnson, and J. D. Tygar, "Completely Asynchronous Optimistic Recovery with Minimal Rollbacks," in *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, Pasadena, California, 1995, pp. 361–371.

[29] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375 – 408, september 2002.