A Consensus-based Fault-Tolerant Event Logger for High Performance Applications

Edson Tavares de Camargo*† and Elias P. Duarte Jr.* and Fernando Pedone‡
*Federal University of Paraná (UFPR), Department of Informatics, Curitiba - Brazil
†Federal Technology University of Paraná (UTFPR), Toledo - Brazil
‡University of Lugano (USI) - Switzerland

***Abstract*—High-performance computing (HPC) systems traditionally employ rollback-recovery techniques to allow fault-tolerant executions of parallel applications. Rollback-recovery based on message logging is an attractive strategy that avoids the drawbacks of coordinated checkpointing in systems with low mean-time between failures (MTBF). Most message logging protocols rely on a centralized event logger to store information (i.e., the determinants) to allow the recovery of an application process. This centralized approach, besides the obvious single point of failure problem, represents a bottleneck for the efficiency of message logging protocols. In this work, we present a fault-tolerant distributed event logger based on consensus that outperforms the centralized approach. We implemented the event logger of MPI determinants using Paxos, a prominent consensus algorithm. Our event logger inherits the Paxos properties: safety is guaranteed even if the system is asynchronous and liveness is guaranteed despite processes failures. Experimental results are reported for the performance of the distributed event logger based both on classic Paxos and parallel Paxos applied to AMG (Algebraic MultiGrid) and NAS Parallel Benchmark applications.**

I. Introduction

High-performance computing (HPC) systems are used to execute complex industrial and scientific simulations, as well as other computing-intensive applications. In these systems, MPI is the *de facto* standard to build parallel and distributed high performance applications that run both on clusters and on dedicated massively parallel processing (MPP) environments [1]. HPC systems, in particular petascale and future exascale systems, are required to cope with increasingly smaller mean time between failures (MTBF) [2]. For example, the Blue Waters petascale system has an average MTBF of 4.2 hours [3]. Future exascale systems should present an even lower MTBF [4], [5].

Most traditional strategies to deal with failures in HPC systems are based on rollback-recovery mechanisms [6], [7]. These strategies allow applications to recover from failures without losing previously computed results. Message logging is a class of rollback-recovery technique that unlike coordinated checkpoint strategies does not require all processes to coordinate to save their state during normal execution and to restart after a single process failure.

Message logging relies on the piecewise deterministic assumption. This assumption states that all nondeterministic events that a process executes can be identified and the information necessary to replay each event during recovery can be logged in tuples called *determinants* [8]. By replaying the determinants in their exact original order, a process can deterministically recreate its pre-failure state. Most message logging protocols suppose that reception events (i.e., message receiving events) are the only possible nondeterministic events in the execution [9]. Consequently, a crucial task in message logging is to reliably save and restore the determinants without penalizing the performance.

The component responsible for reliably logging determinants is the *event logger*. The event logger receives the determinants from the application processes, stores them locally, and notifies the application processes. Previous works based on message logging typically assume that the event logger is a centralized entity (e.g., [10], [11], [12]), and thus it cannot tolerate failures. Indeed, the failure of the event logger would bring the execution to a halt as application processes would no longer be able to save the determinants.

The main goal of this paper is to propose a fault-tolerant event logger that has performance comparable to or better than a centralized event logger. In particular, our replicated event logger does not require extra system resources (i.e., physical nodes) in comparison with a centralized event logger and can tolerate a configurable number of failures. When configured to tolerate a single failure, our consensus-based event logger needs the same number of messages and communication steps (i.e., network delays) to log a determinant as a centralized event logger. We also show in the paper that the myth that fault tolerance introduces overheads is not completely unfounded since the indiscriminate use of existing fault-tolerance techniques can indeed lead to expensive solutions.

We implemented two fault-tolerant event loggers based on the Paxos algorithm [13]. One is based on classic Paxos and the other on a configuration we call parallel Paxos. We conducted a number of experiments comparing them to a centralized event logger. We evaluated the performance of our event logger implementations using the LU and MG kernels from the NAS Parallel Benchmarck (NAS-PB) and the Algebric MultiGrid (AMG) application. Our results show that the replicated event logger based on parallel Paxos consistently outperforms a centralized event logger while providing configurable fault tolerance.

The rest of the paper is organized as follows. Section II briefly overviews rollback-recovery, including message log-

ging, and the event logger. Section III reviews the Paxos algorithm and presents our consensus-based event loggers. Section IV presents our implementations of the event logger and experimental results. Section V describes related work and Section VI concludes the paper.

II. A Rollback-Recovery Primer

Rollback-recovery techniques are often used to provide fault tolerance to HPC applications so that they can restart from a previously saved state [2], [9]. Rollback-recovery assumes a distributed system that is a collection of application processes that communicate through a network and have access to stable storage that survives failures [8]. Processes save recovery information periodically on stable storage during their failure-free execution. After the occurrence of a failure, the process that failed uses the recovery information to restart its computation from a past state. The recovery information includes at least the state of the participating processes, called *checkpoints*. Some protocols may also include the logging of nondeterministic events, encoded in tuples called *determinants*. A consistent system state is one in which if the state of a process includes the receipt of a message, then the state of the sender includes the transmission of that message [14]. A set of checkpoints that corresponds to a consistent state is called a *recovery line*. The main goal of a rollback-recovery protocol is to restore the system back to the most recent recovery line after a failure. Rollback-recovery protocols can be either checkpoint-based or log-based, as described next.

A. Checkpoint-based Rollback Recovery

In the checkpoint-based approach, recovery is performed only through checkpoints and is classified as either coordinated or uncoordinated. Checkpoints that are taken independently by each process, without any global coordination, are known as uncoordinated. Uncoordinated checkpointing has the advantage that processes create checkpoints when it is most convenient for them (independently of other processes) thereby reducing the communication overhead. However this approach may never lead the application to a consistent state. Checkpoints that lead to an inconsistent state are useless and must be discarded [8], [15].

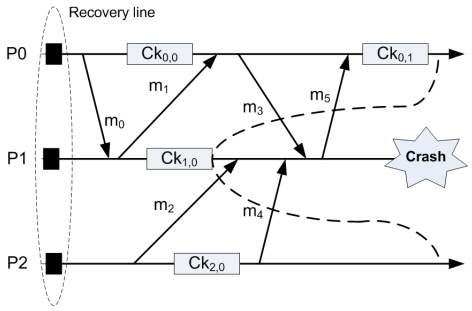

Figure 1. Uncoordinated checkpointing and a recovery line.

Figure 1 shows a scenario in which the most recent set of checkpoints (i.e., $Ck_{0,1}$, $Ck_{1,0}$, and $Ck_{2,0}$) results in no recovery line. This set of checkpoints does not correspond to a consistent state because message m_5 is received by P_0 but not sent by P_1 — in such a case, m_5 is called an *orphan message* and P_0 an *orphan process*. P_0 is then forced to roll back to an earlier checkpoint (i.e., $Ck_{0,0}$). Because messages m_1 and m_2 have to be retransmitted, the recovery line corresponds to the initial state of the application. This is known as the *domino effect*. Uncoordinated checkpointing is susceptible to the domino effect.

Coordinated checkpointing avoids the domino effect by synchronizing the checkpoints of each process in order to save a system-wide consistent state [15]. Although coordinated checkpointing is relatively easy to implement, its execution is quite expensive, as processes need to coordinate. Furthermore during recovery every process has to roll back to its last checkpoint even if a single process fails.

B. Log-based Rollback Recovery

Log-based approaches, or simply "message logging", use both checkpoints and logging of nondeterministic events to avoid the drawbacks of both uncoordinated and coordinated checkpointing. Message logging protocols assume the application is piece-wise deterministic [15]. This assumption asserts that all nondeterministic events executed by a process can be identified and the information necessary to replay each event during recovery can be logged in determinants. An *event* corresponds to a computational or communication step of a process. Most message logging protocols assume that message reception is the only nondeterministic event. Message logging avoids the domino effect of uncoordinated checkpointing by saving all received messages. For example, in Figure 1, messages m_2, m_4, and m_3 received by process P_1 must be saved, as well as the determinants that contain the order of the messages. Upon the recovery of process P_1, only P_1 rolls back. Thus, the state of P_1 will eventually be the same as it was before the failure, as messages m_2, m_4, and m_3 are received again in the same order.

Depending on how determinants are logged, message logging protocols can be pessimistic, optimistic or causal [8]. In pessimistic logging, a process first stores the determinant of a nondeterministic event (e.g., in remote storage) before delivering the message. Despite the fact that pessimistic logging simplifies recovery and garbage collection, it presents an overhead on failure-free scenarios: the application has to wait for the determinant to be stored in order to proceed. In optimistic logging, processes log determinants asynchronously, thereby reducing the overhead. However, optimistic protocols allow orphan processes to be created due to failures and lead to more expensive recovery. Causal logging protocols try to combine the advantages of pessimistic and optimistic logging [12]: low performance overhead on failure-free scenarios and no orphans processes.

However, causal logging needs to piggyback the determinants on each application message until these determinants are safely logged.

Usually, in message logging approaches the determinant of every received message is logged. However, it is possible to reduce the overall number of logged messages by identifying which events are deterministic and which are nondeterministic [16]. An event is deterministic when from the current state there is only one possible outcome state for the event. If an event can result in several different states, then it is nondeterministic. Message receptions with an explicitly identified sender are deterministic events and do not need to be logged; if the source is left unspecified then message receptions are nondeterministic. For example, a nondeterministic event occurs in MPI when the receiving process uses the tag `MPI_ANY_SOURCE` in `MPI_Recv`. As stated in [17], several MPI applications contain only deterministic communication events and some protocols were proposed for this class of applications [7], [18], [19], [20]. However, many important MPI applications are nondeterministic. Furthermore, programmers usually include nondeterministic in the code to improve performance.

Message logging usually relies on the sender-based approach [21]. In this approach, during normal operation the payload of every outgoing message is saved at the sender. The receiver is required only to log the determinant, describing the message's delivery event.

C. The Event Logger

The event logger plays an important role in message-logging protocols [12]. It receives the determinants from the application processes, stores them locally, and notifies the application processes after determinants are stored. The performance of the event logger has a major impact on the efficiency of message-logging protocols as showed in [11], [12], [22], and many protocols implement the event logger as a centralized (i.e., non-replicated) component [9], [23].

It has also been shown that the event logger can improve the performance of causal message logging protocols. In these protocols the logger helps to decrease the size of the causality information that is piggybacked in the protocol messages. However, in order to achieve this reduction of piggybacking all processes have to be notified about every event information that is saved on reliable storage. Thus a centralized event logger may become a bottleneck, as the number of processes grows.

III. Consensus and message logging

Consensus is a fundamental abstraction in fault-tolerant distributed computing. In this section, we review the consensus problem, present Paxos, one of the most prominent consensus algorithms, and discuss how to efficiently implement a fault-tolerant event logger with Paxos.

A. Consensus and State Machine Replication

Consensus can be used to build a highly available event logging service using the state machine replication approach [24]. State machine replication regulates how commands must be propagated to and executed by the replicas in order for the service to be consistent. In our particular case, the commands are requests to save a determinant, propagated to and executed by replicas of the event logger. Command propagation has two requirements: (i) every non-faulty replica must receive every command and (ii) no two replicas can disagree on the order of received and executed commands. If command execution is deterministic, then replicas will reach the same state and produce the same output upon executing the same sequence of commands.

Intuitively, consensus captures the command propagation requirements of state machine replication. More precisely, consensus is defined by three abstract properties: (a) If a replica decides on a value, then the value was proposed by some process (*validity*). (b) No two replicas decide differently (*agreement*). (c) If a non-faulty process proposes a value, then eventually all non-faulty replicas decide some value (*termination*). From the requirements of state machine replication and the guarantees provided by consensus, it should be clear that state machine replication can be implemented as a series of consensus instances, where the i-th consensus instance decides on the i-th command (or batch of commands) to be executed by the replicas [25].

State machine replication and consensus provide a principled approach to ensuring that replicas are consistent despite failures. This approach should not be overlooked since ad hoc solutions to replication must face subtle impossibilities in the design of distributed systems subject to process failures [26].

B. The Paxos Protocol

Paxos is a fault-tolerant consensus algorithm designed for state machine replication [25]. Paxos has important characteristics: it is safe under asynchronous assumptions, live under weak synchronous assumptions, ensures progress with a majority of non-faulty processes, and assumes a crash-recovery failure model.

Paxos distinguishes the following roles that a process can play: *proposers*, *acceptors* and *learners*. Proposers propose a value, acceptors choose a value, and learners learn the decided value. A single process can assume any of those roles, and multiple roles simultaneously. Paxos is resilience-optimum [27]: to tolerate f failures it requires $2f + 1$ acceptors—that is, to ensure progress, a quorum of $f + 1$ acceptors must be non-faulty.

An instance of Paxos proceeds in two phases: during the first phase, a proposer selects a unique round number and sends a *prepare* request to a quorum of acceptors. Upon receiving a prepare request with a round number bigger than any round the acceptor previously received, the

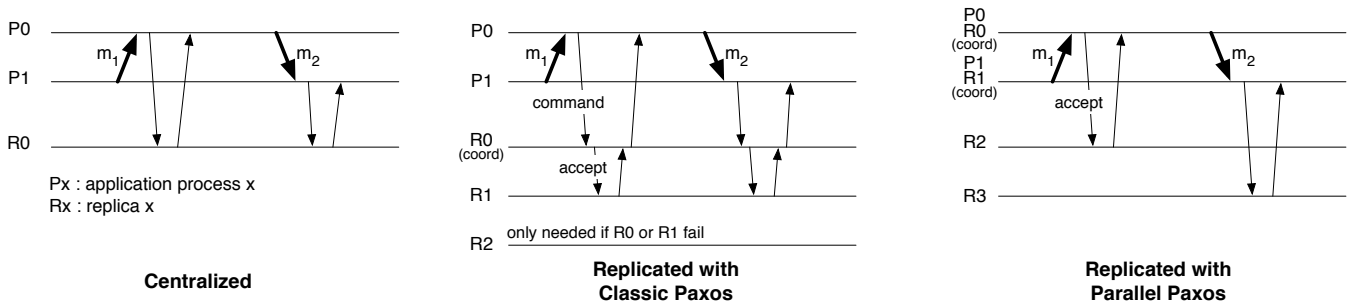

Figure 2. Three implementations of an event logger. The centralized approach has a single event logger (R_0) and thus cannot tolerate any failures. The Paxos-based approaches can tolerate one failure (i.e., $f = 1$). Paxos coordinators already executed the first phase of the protocol and can proceed with the second phase upon receiving a command.

acceptor responds to the proposer promising that it will reject any future requests with smaller round numbers. If the acceptor already accepted a command for the current instance (explained next), it will return this command to the proposer, together with the round number received when the command was accepted. When the proposer receives answers from a quorum of acceptors, it proceeds to the second phase of the protocol.

In the second phase, the proposer selects a command according to the following rule. If no acceptor in the quorum of responses accepted a command, the proposer can select a new command for the instance; however, if any of the acceptors returned a command in the first phase, the proposer chooses the command with the highest round number. The proposer then sends an *accept* request with the round number used in the first phase and the command chosen to a quorum of acceptors. When receiving such a request, the acceptors acknowledge it by sending a message to the coordinator and learners, unless the acceptors have already acknowledged another request with a higher round number. When a quorum of acceptors accepts a command consensus is reached.

If multiple proposers simultaneously execute the procedure above for the same instance, then no proposer may be able to execute the two phases of the protocol and reach consensus. To avoid scenarios in which proposers compete indefinitely, a *coordinator* process can be chosen. In this case, proposers submit commands to the coordinator, which executes the first and second phases of the protocol. If the coordinator fails, another process takes over its role. Paxos ensures consistency despite concurrent coordinators and termination in the presence of a single coordinator.

A coordinator can optimize performance by executing the first phase of the protocol for a batch of instances before it receives any commands [13]. This is possible because the coordinator only sends commands in the second phase of the protocol. With this optimization, a command can be chosen in three communication steps: the message from the proposer to the coordinator, the accept request from the coordinator to the acceptors, and the response to this request from the acceptors to the coordinator and learners.

C. Consensus-based Message Logging

We now propose two protocols based on Paxos to render the event logger fault-tolerant: Classic Paxos and Parallel Paxos. Similarly to a centralized event logger, our protocols log nondeterministic events only; the message payload is saved by the sender. A determinant contains the sender of a message, the message identifier, and the message receiving order. Periodically, each process performs a checkpoint in order to save its state.

Our first protocol, Classic Paxos, is based on classic state machine replication. Application processes are proposers and the event logger replicas are acceptors and learners. Every application process submits commands to the coordinator, a process among the acceptors, to log determinants. The coordinator receives commands, executes Paxos to log the commands in a quorum of replicas, and sends replies to the application process (see Figure 2). In "good executions" (i.e., in the absence of process failures) a determinant is logged after four communication steps and $2f + 2$ point-to-point message exchanges. By contrast, a centralized event logger can log events after two communication steps and two message exchanges.

In our second protocol, Parallel Paxos, we assign a separate sequence of Paxos executions to each application process. This means that each process has its set of replicas, which allows important optimizations. First, since each process has its own sequence of Paxos executions, the process does not compete with other processes in executions of Paxos and therefore, there is no need for a coordinator; in good executions, the process is the only proposer in its sequence of Paxos. Second, by using different sets of replicas, performance is no longer capped by what the coordinator and the acceptors can handle. Third, we can now co-locate the application process and the acceptor-coordinator in the same process. In good runs, this scheme can log a determinant after two communication steps and $2f$ messages.

An event logger implemented with Parallel Paxos presents the same number of communication steps as a centralized event logger, while tolerating a configurable number of failures and scaling performance. When configured to tolerate one failure ($f = 1$), it exchanges the same number of messages per logging operation as the centralized logger. Moreover, to save resources, "free acceptors" (i.e., acceptors not collocated with application processes) can be placed in the same physical node. For example, a single node can host all free acceptors, i.e. Parallel Paxos can use the same amount of nodes required by a centralized strategy.

Upon recovering from a failure, an application process must retrieve all its logged determinants. With the centralized approach, the application process contacts the event logger. With the replicated approaches, this is done by contacting a quorum of acceptors.

IV. Evaluation

In this section we describe an implementation of the proposed consensus-based event logger and present experimental results, including a comparison with the traditional centralized alternative. Results are presented for the execution of three MPI applications: AMG[1] (Algebraic Multigrid Solver), LU (Lower-Upper Gauss-Seidel solver) and MG (Multi-Grid on a sequence of meshes). Both LU and MG are on the NAS parallel benchmarks version 3.2.[2] The applications were executed through OpenMPI[3] version 1.10. The experiments were conducted on a dedicated cluster that consists of 40 nodes each with two Intel(R) Quad-Core Xeon L5420 2.5 GHz processors and 8 Gbytes of RAM interconnected on a Gigabit Ethernet network.

We intercept MPI primitives using the MPI standard profiling interface (PMPI) [28]. If the interceptor detects a nondeterministic event, as defined in [16], it builds a determinant related to the event and makes a submission to the event logger. We implemented three event loggers: a traditional centralized logger, a distributed replicated logger based on Classic Paxos, and the distributed replicated logger based on Parallel Paxos (all described in Section III-C). The Parallel Paxos event logger can be configured to log messages synchronously or asynchronously. In the synchronous mode, after submitting a determinant, an application process waits for an acknowledgement from the event logger before submitting the next determinant; in asynchronous mode the application process can submit multiple determinants before it receives acknowledgments from the event logger. Unless stated otherwise, our experiments use the synchronous mode. The interceptor and event loggers were implemented in C using the libevent version 2.022.[4] We used the Paxos library libpaxos version 3.[5]

The centralized event logger is hosted on a dedicated node. In Classic Paxos, a coordinator was deployed on a dedicated node while three acceptors (i.e., $f = 1$) were deployed each on a single node. There were also three learners, each one colocated with an acceptor. The learners are responsible for replying to the MPI processes as soon as a determinant is stored. In Parallel Paxos, each sequence of Paxos executions uses three acceptors (i.e., $f = 1$), each one deployed on a dedicated node. In Parallel Paxos each MPI process is both a proposer and a learner. Acceptors can be configured to store commands (i.e, determinants) on disk or in memory. In all experiments, the centralized event logger and the acceptors log values in main memory. We justify this choice by the fact that persistent memory technologies such as non-volatile RAM (NVRAM) and battery-backed memory are increasingly popular.

A. The Event Logger

To evaluate the performance of the event logger alone, we built a simple MPI application where each process submits determinants to the logger in a closed loop, i.e. a process only submits a new determinant to the logger after it receives a response acknowledging that the previously submitted determinant has been logged. Since application processes do not communicate among themselves in these experiments, determinants are fixed-content 50-byte messages.

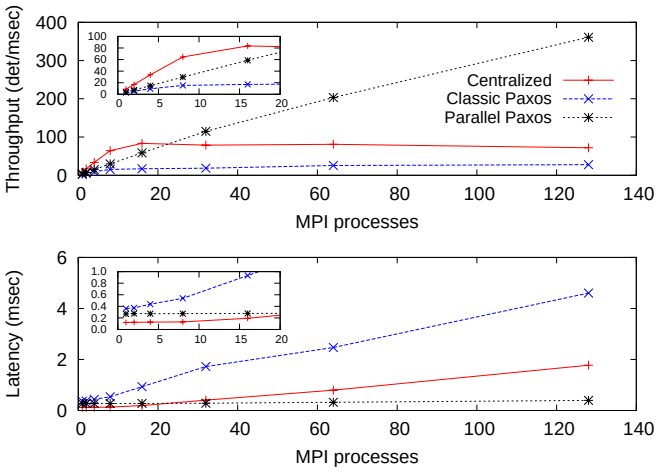

Figure 3. Throughput and latency for the three event logger approaches.

Figure 3 shows the throughput (in logged determinants per millisecond) and latency (in milliseconds) when we increased the number of MPI processes up to 128. The centralized event logger reaches the maximum throughput of about 83 determinants per millisecond with 16 processes. Classic Paxos reaches a maximum throughput of 28 determinants logged per millisecond with a latency of

[1]https://codesign.llnl.gov/amg2013.php
[2]https://www.nas.nasa.gov/publications/npb.html
[3]http://www.open-mpi.org
[4]http://libevent.org/
[5]https://bitbucket.org/sciascid/libpaxos

Benchmark	Number of application processes			
	16	**32**	**64**	**128**
AMG	0.05%	0.02%	0.05%	0.04%
LU class C	0.63%	0,63%	0.63%	0.63%
LU class D	0.25%	0.25%	0.25%	0.25%
MG class C	99.81%	99.81%	99.81%	99.80%
MG class D	99.93%	99.93%	99.93%	99.93%

Table I
PERCENTAGE OF NONDETERMINISTIC EVENTS OF THE BENCHMARKS.

4.6 milliseconds with 128 MPI processes. Parallel Paxos never saturates in these experiments: throughput increases proportionally to the number of processes and the latency remains approximately constant, below 4 milliseconds. With 128 processes, Parallel Paxos has 5 times the throughput of the centralized scheme and 13 times the throughput of Classic Paxos, with much lower latency.

B. AMG

AMG is a parallel algebraic multigrid solver for linear systems which can be classified as a nondeterministic application that employs "any-source" receptions and nondeterministic deliveries. All calls to `Iprobe` use the `any_source` tag and only one call to `Recv`, among many, uses the `any_source` tag. AMG also has calls to the `Test` and `Testall` primitives. Table I shows the percentage of nondeterministic events in AMG in executions with 16, 32, 64 and 128 MPI processes. We set AMG parameters rx, ry, and rz equal to 40. Although during the execution there was a large number of `Iprobe`, `Test` and `Testall` invocations, a determinant for an `Iprobe` with the `any_source` tag is only created when the message is ready to be received. Similarly, for `Test` and `Testall` invocations, we count the number of invocations but only submit the determinant to the event logger when the MPI message related to `Test` or `Testall` is ready to be delivered.

Figure 4 (left) presents results for the AMG application, including the distributed and replicated event loggers. Although all strategies introduce an overhead, with 64 and 128 processes Classic Paxos increased the duration of the original application by approximately 3.8%, the worst-performing technique. The centralized scheme presented an overhead of approximately 2% for 16, 32 and 64 processes. Parallel Paxos presented the lowest overhead, below 1.3% for all configurations.

Figure 4 (right) shows the number of determinants logged per millisecond considering both the synchronous and asynchronous Parallel Paxos modes, for 16, 32, 64 and 128 MPI processes. In the asynchronous mode, application processes never wait for the logger; thus, this case provides an upper bound for the performance of the event logger. As it can be seen, log requests are not uniformly distributed over time. For the case with 128 processes, between time instants 80s and 90s a peak can be distinguished, that reaches approximately 34 determinants per millisecond in the synchronous mode and 66 determinants per millisecond in the asynchronous mode. These results help understand how the overhead of Parallel Paxos is distributed over time.

C. LU and MG

LU and MG are kernels of the NAS parallel benchmarks. As pointed out in [16], only MG and LU among the NAS-PB kernels generate nondeterministic events. Furthermore, in both the only nondeterministic events are "any-source" receptions; there are no nondeterministic deliveries such as `Test`. We assess classes C and D of the kernels in deployments with 16, 32, 64 and 128 MPI processes. LU contains both `Recv` and `Irecv` primitives. The last one is used with the `any_source` tag. MG receives all its messages through `MPI_Irecv` with the `any_source` tag.

As shown in Table I, the total number of events logged in LU is less than 1% of the total of all its receptions in classes C and D. The MG kernel however has almost 100% of nondeterministic events among its receptions. Although both AMG e MG solve similar problems, the reason for much more nondeterministic events in MG is its implementation. Unlike MG, AMG does not receive all its messages through `MPI_Irecv` with the `any_source` tag. This illustrates the fact that nondeterminism is often a programmer's choice (e.g., to boost performance), rather than a requirement coming from the problem being solved.

From our experimental evaluation, we concluded that logging determinants using any of the three event logging strategies presents nearly no overhead when logging events of classes C and D of the LU kernel. This is somewhat surprising since logging introduces some overhead in the AMG application and both classes C and D of the LU benchmark contain a higher percentage of nondeterministic events than AMG contains (see Table I). Figure 5 (left) shows the results for class C of the LU kernel. By inspecting the number of determinants logged during the execution in Figure 5 (right), we notice that determinants are more uniformly distributed in LU class C than in AMG and they happen at a rate that is within the limits the event logger can sustain (see Section IV-A). The difference between class C and D of the LU is that the last one has longer duration and lower throughput. As a consequence, the event logger never becomes an execution bottleneck in LU.

On the contrary, the logging of determinants introduces a considerable overhead to MG classes C and D (Figures 6 and 7, respectively). In class C, while Classic Paxos presents an overhead of more than 125% and 200% for 64 and 128 processes, the overhead of the centralized event logger is below 31% and 55% for 64 and 128 processes, respectively. Parallel Paxos sports even lower overheads: 17,71% and 24,26% for 64 and 128 processes, respectively. The results for MG class D show a similar trend, with Parallel Paxos outperforming both the two other techniques.

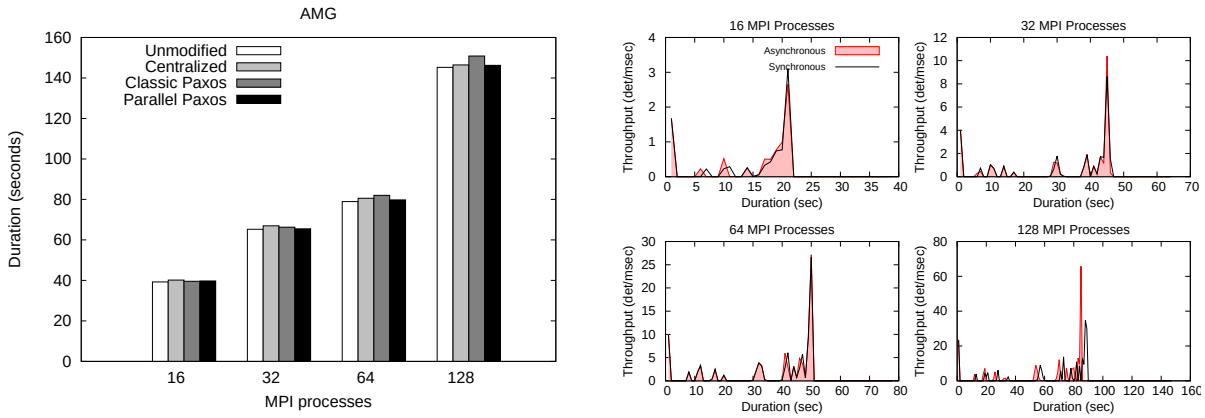

Figure 4. AMG with and without logging nondeterministic events. AMG througput using synchronous and asynchronous parallel Paxos.

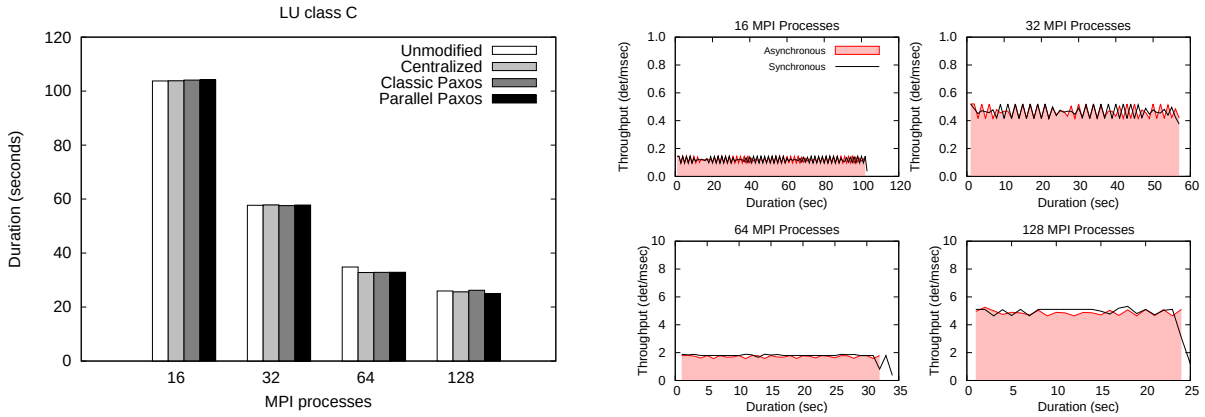

Figure 5. LU class C performance and throughput with and without logging nondeterministic events.

Table II shows the overheads for all logging techniques and configurations of MG classes C and D. The MG kernel is highly communication-bound and all its receive events use the `any_source` tag. As the number of application processes increases, the event logger reaches its limits with the centralized and the Classic Paxos strategies. Parallel Paxos is able to scale performance by distributing the load among the various series of Paxos.

	Proc	Centralized	Classic Paxos	Parallel Paxos
MG class C	16	0.03%	11.81%	1.43%
	32	7.39%	43.61%	5.53%
	64	30.52%	125.78%	17.71%
	128	54.38%	204.86%	24.26%
MG class D	16	0.60%	0.29%	0.42%
	32	0.69%	0.45%	0.72%
	64	4.60%	25.43%	2.24%
	128	9.48%	41.91%	2.93%

Table II
OVERHEAD OF AN EVENT LOGGER WITH CENTRALIZED, CLASSIC PAXOS, AND PARALLEL PAXOS IN MG CLASSES C AND D.

Figures 6 and 7 also show the rate of logged determinants per millisecond for the synchronous and asynchronous Parallel Paxos-based event logger. The throughput of the synchronous mode is close to the asynchronous mode for MG class C with 16, 32 and 64 processes. For 128 processes, the asynchronous mode presents a throughput that is higher than that of the synchronous mode and finishes approximately 1 second earlier. MG class D has lower throughput than MG class C. The throughputs of both synchronous and asynchronous Parallel Paxos are very similar. In all configurations, both the synchronous and the asynchronous modes display a uniform rate over time.

V. RELATED WORK

In this section we briefly review some relevant message logging protocols and the role that the event logger plays in different systems.

There are three message logging protocols with uncoordinated checkpointing within the MPICH-V project [9]. Two of those protocols are pessimistic and one is causal. The MPICH-V1 is a pessimistic protocol for high volatility and heterogeneous resources, such as desktop grids. MPICH-V makes use of a remote stable component called Channel

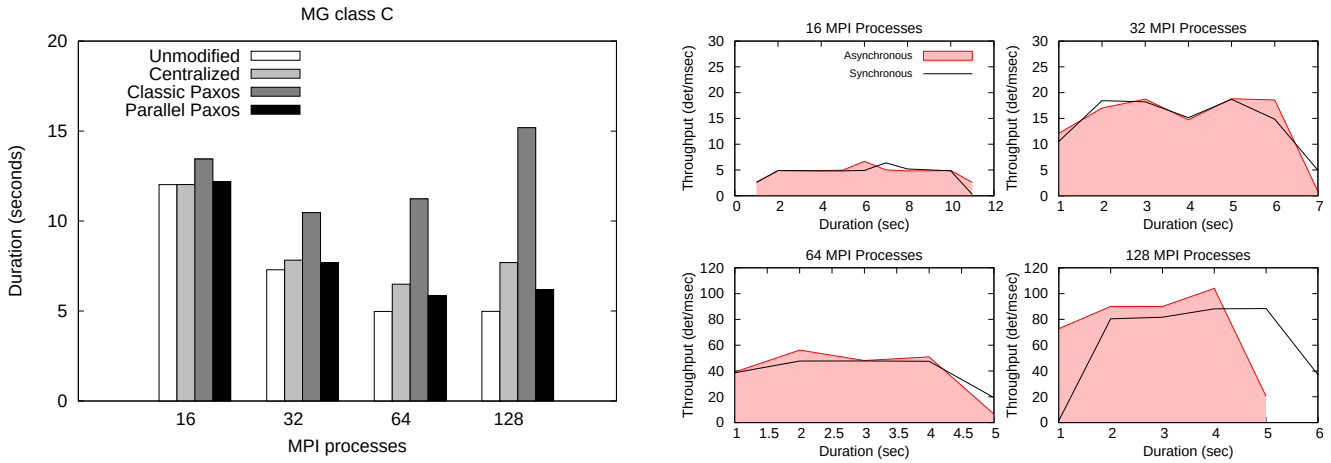

Figure 6. Performance comparison: MG class C application. Performance comparison: MG class D application.

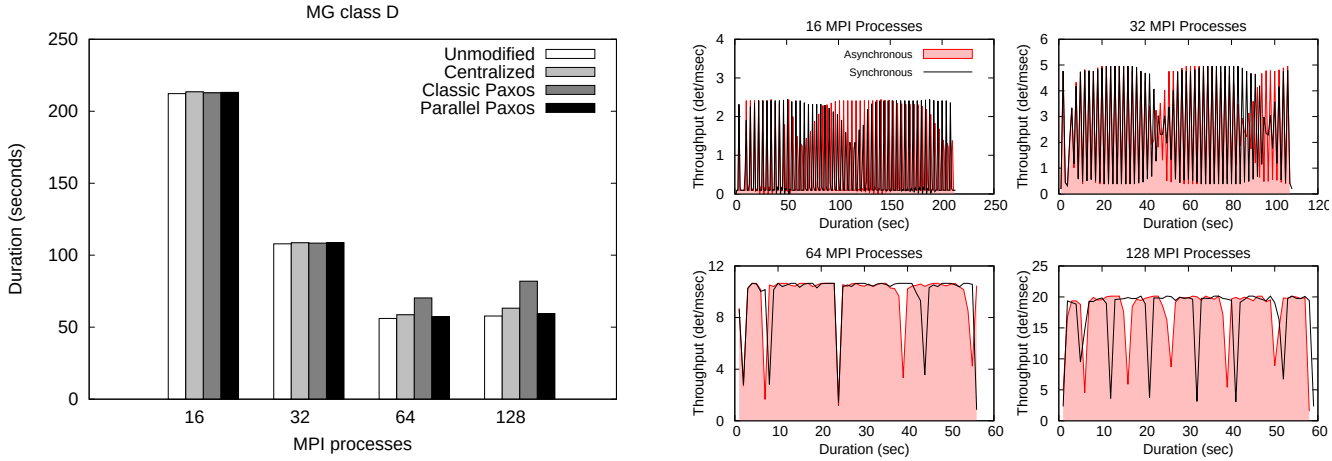

Figure 7. MG with and without logging nondeterministic events. MG throughput using synchronous and asynchronous parallel Paxos.

Memory (CM) that stores the payload and the order of receptions of the MPI messages. Every process first sends the message to the CM of the receiver. Then, the receiver makes a request for the message from its own CM. Although there is one CM for each process, they do not support failures. MPICH-V2 [23] is a pessimistic protocol designed for large clusters. MPICH-V2 relies on sender-based approach. Processes communicate directly. Instead of using CMs, MPICH-V2 employs event loggers that are used as remote stable storage. When a process receives a message it sends the determinant to an event logger. MPICH-V2 assumes that the event logger is reliable.

In [29] a coordinated checkpointing strategy based on the Chandy-Lamport algorithm is compared with MPICH-V2 using uncoordinated checkpoint. The main difference between the two approaches is related to the occurrence and frequency of faults. When the number of faults increases, the restart cost of the coordinated approach compensates for the overhead of pessimistic message logging.

In [12], the authors investigate the benefits of using an event logger for causal message logging protocols. They implemented and evaluated three protocols: Manetho, LogOn and Vcausal. These protocols are evaluated with and without an event logger. The authors show that the presence of the event logger has a major impact on the following four performance metrics: piggybacking computation cost, piggyback size, application performance and fault recovery; this is independent of the causal message logging employed. The event logger is assumed to be reliable. The authors also highlight that using only one event logger for consistency becomes a bottleneck as the number of processes grow. They also state that it is necessary to investigate how to distributed the logging of events among multiple event loggers.

The work in [10] experimentally compares a pessimistic and an optimistic message logging protocols considering the refinement of the message logging model proposed in [16]. By implementing these refinements in a protocol called `Vprotocol` inside the Open MPI library, the number of

messages sent to the event logger decreases considerably. In the `Vprotocol` the event logger is not fault-tolerant and is implemented as a special process made available to the application from outside the main MPI group, that is, the `MPI_COMM_WORLD`. Currently the `Vprotocol` is not available in the Open MPI library. We implemented the same refinements proposed in [16]. Unlike `Vprotocol`, our event logger is not an MPI process.

In [11] the O2P active optimistic message logging protocol is proposed. In this protocol, the event logger is implemented as a single-threaded MPI process that can handle asynchronous communication. Like our proposed event logger, the O2P event logger is also started separately from the application processes. Application processes connect to the event logger when they start logging determinants. However, in O2P the event logger is assumed to be reliable. The experiments with a large number of processes and a high communication rate indicate that the event logger is the system bottleneck. A distributed event logger for O2P is proposed in [22]. The event logger takes advantage of multi-core processors to be executed in parallel with the application processes. Despite the fact that the protocol offers a distributed way to save determinants, the entire solution fails if an acknowledgement is not received by the sender. That is, the fault tolerance of the solution is not guaranteed. Our proposed event logger saves the determinants in the acceptor memory. The Paxos protocol ensures progress with a majority of non-faulty processes and assumes a crash-recovery failure model.

A hybrid protocol that combines coordinated checkpointing and optimistic message logging is proposed in [30]. It uses additional nodes that can work both as event loggers and as spares to be swapped in when nodes fail. Coordinated checkpointing assists message logging by limiting the log sizes and avoiding making the application return to the initial state in case it reaches an inconsistent state. Message logging in turn is capable of avoiding restarting all processes in case of failures most of the time. The work assumes the presence of a Reliability, Availability, and Serviceability (RAS) system for detecting process faults and restarting processes on spare nodes. If an event logger fails, the application continues to execute. If the application reaches a global checkpoint, the RAS system uses a spare node to assign a new external logger and informs every process about it. However, if a node needs the recovery logs from the failed external logger, the application must return to the last coordinated checkpoint. Although the event logger does not use stable storage, it is not distributed and requires additional resources in case of failures. Using an optimistic protocol decreases the overhead but may create orphan processes.

There have been several attempts to reduce the overhead of message logging. Some protocols take into account only send deterministic applications [17]. Examples include [7], [18], [20]. Although they do not use an event logger, these protocols still need to log a subset of the application messages in the volatile memory of each process. These works are limited to send-deterministic applications.

A related work that uses a consensus protocol to build fault-tolerant HPC applications is [31] which proposes an agreement algorithm implemented in the User Level Failure Mitigation (ULFM) API [32]. ULFM extends the MPI specification by providing a well-defined flexible mechanism that allows applications and libraries to handle multiple types of faults. The algorithm is designed for an MPI environment based on the fail-stop assumption, which is stronger than our assumptions. Previously, in [33], [34] agreement algorithms were proposed to be used in a similar context. To the best of our knowledge our work is the first that applies Paxos for creating a distributed and fault-tolerant event logger for HPC.

VI. Conclusion

In this work we presented a fault-tolerant and distributed event logger based on consensus for HPC applications. The event logger is the component responsible for reliably logging determinants and its performance can represent a significant impact on the efficiency of message logging protocols. We implemented two fault-tolerant event loggers based on the Paxos algorithm. By using Paxos, our event loggers guarantee safety even if the system is asynchronous and liveness despite processes failures. Our first protocol is based on classic state machine replication. In our second protocol, which we call Parallel Paxos, we assign a separate sequence of Paxos executions to each application process. We assessed experimentally the performance of a centralized event logger and our two event loggers based on consensus. Besides evaluating the event loggers by themselves, we used three MPI applications to evaluate their performance: AMG, MG and LU. Results of all experiments show that the event logger based on Parallel Paxos always outperformed the centralized approach in terms of both the execution time and the throughput in terms of the number of determinants logged per millisecond.

As future work, Parallel Paxos can be extended to be used for a causal message logging protocol. In this case it is possible to use a gossip protocol to spread the logged determinants between all processes. The implementation of a recovery protocol using the nondeterministic events stored on the Parallel Paxos event logger is also left as future work.

References

[1] G. E. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *Recent advances in PVM and MPI*, ser. LNCS, 2000.

[2] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for hpc systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[3] C. Di Martino, Z. Kalbarczyk, R. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *DSN*, 2014.

[4] M. S. et. al., "Addressing failures in exascale computing," *International Journal of HPC Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[5] F. Cappello, A. Geist, B. Gropp, L. V. Kalé, B. Kramer, and M. Snir, "Toward exascale resilience," *International Journal of HPC Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[6] D. Tiwari, S. Gupta, and S. Vazhkudai, "Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems," in *DSN*, 2014.

[7] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic MPI applications," in *IPDPS*, 2011.

[8] Elnozahy, Alvisi, Wang, and Johnson, "A survey of rollback-recovery protocols in message-passing systems," *CSURV: Computing Surveys*, vol. 34, 2002.

[9] A. Bouteiller, T. Hérault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V project: A multiprotocol automatic fault-tolerant MPI," *International Journal of HPC Applications*, vol. 20, no. 3, pp. 319–333, 2006.

[10] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra, "Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure, recovery," in *Cluster*, 2009.

[11] T. Ropars and C. Morin, "Active optimistic message logging for reliable execution of MPI applications," in *Euro-Par*, 2009.

[12] A. Bouteiller, B. Collin, T. Herault, P. Lemarinier, and F. Cappello, "Impact of event logger on causal message logging protocols for fault tolerant mpi," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005, pp. 97–97.

[13] Lamport, "Paxos made simple," *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, vol. 32, 2001.

[14] M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[15] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge, UK: Cambridge University Press, Mar. 2011.

[16] A. Bouteiller, G. Bosilca, and J. Dongarra, "Redesigning the message logging model for high performance," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 16, pp. 2196–2211, 2010.

[17] F. Cappello, A. Guermouche, and M. Snir, "On communication determinism in parallel HPC applications," in *ICCCN*, 2010.

[18] A. Guermouche, T. Ropars, M. Snir, and F. Cappello, "HydEE: Failure containment without event logging for large scale send-deterministic MPI applications," in *IPDPS*, 2012.

[19] A. Lefray, T. Ropars, and A. Schiper, "Replication for send-deterministic MPI HPC applications," in *FTXS Workshop at HPDC*, 2013.

[20] T. Ropars, T. V. Martsinkevich, A. Guermouche, A. Schiper, and F. Cappello, "SPBC: leveraging the characteristics of MPI HPC applications for scalable checkpointing," in *SC*, 2013.

[21] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging," in *FTCS*, 1987.

[22] T. Ropars and C. Morin, "Improving message logging protocols scalability through distributed event logging," in *Euro-Par*, 2010.

[23] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *SC*, 2003.

[24] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 3, p. 299, Dec. 1990.

[25] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst*, vol. 16, no. 2, pp. 133–169, 1998.

[26] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty processor," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[27] L. Lamport, "Lower bounds for asynchronous consensus," *Distributed Computing*, vol. 19, no. 2, pp. 104–125, 2006.

[28] MPI Forum, "Document for a standard message-passing interface 3.1," University of Tennessee, http://www.mpi-forum.org/docs/mpi-3.1, Tech. Rep., 2015.

[29] P. Lemarinier, A. Bouteiller, G. Krawezik, and F. Cappello, "Coordinated checkpoint versus message log for fault tolerant MPI," *International Journal of High Performance Computing and Networking*, pp. 146–155, 2006.

[30] R. Riesen, K. Ferreira, D. D. Silva, P. Lemarinier, D. Arnold, and P. G. Bridges, "Alleviating scalability issues of checkpointing protocols," in *SC*, 2012.

[31] T. Herault, A. Bouteiller, G. Bosilca, M. Gamell, K. Teranishi, M. Parashar, and J. Dongarra, "Practical scalable consensus for pseudo-synchronous distributed systems," in *SC*, 2015.

[32] W. Bland, A. Bouteiller, T. Hérault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *International Journal of HPC Applications*, vol. 27, no. 3, pp. 244–254, 2013.

[33] D. Buntinas, "Scalable distributed consensus to support MPI fault tolerance," in *IPDPS*, 2012.

[34] J. Hursey, T. Naughton, G. Vallée, and R. L. Graham, "A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI," in *EuroMPI*, 2011.