Distributed Snapshot for Rollback-Recovery with One-Sided Communications

Franck Butelle
Franck.Butelle@lipn.univ-paris13.fr

Camille Coti
Camille.Coti@lipn.univ-paris13.fr

LIPN, CNRS-UMR7030, Université Paris 13, F-93430 Villetaneuse, France

***Abstract*—Traditional interprocess communication requires cooperation and synchronization between sender and receiver. The One-sided communication model is a new way and very promising model: processes can directly read or write in the memory of another process. In such a model, fault tolerance is a challenging problem. In this paper, we present algorithms to be able to do rollback-recovery based on global snapshot in a distributed system with one-sided communications.**

***Keywords*-One-sided communications; distributed snapshot; Remote Direct Memory Access**

I. Introduction

In high-performance computing, the number of cores in each system has been increasing dramatically since the last decade. In the latest Top 500 ranking released in November 2017, 499 machines feature more than 10^4 cores [1]. At this scale, failures are a significant issue and cannot be ignored: even with very reliable individual components, the Mean Time Between Failures (MTBF) of the whole system is only a few hours [2]. A study made on the pre-production phase of Blue Waters showed that there was a failure every 4.2 hours, and 58.3% of these failures caused (single- or multi-) node unavailability [3]. Hence, fault tolerance has been identified as a key challenge for programming exa- and petascale machines [4], [5], [6].

Besides, while MPI two-sided communications has been the *de facto* standard for programming parallel applications for the last 25 years, other communication models are used nowadays in order to design more scalable applications and to face the relatively slow communications compared to the computation speed of individual nodes [7]. One-sided communications have gained large attention, with the development of one-sided MPI communications and their integration to the MPI2 and MPI3 standards [8], [9] and other models such as OpenSHMEM [10]. One-sided communications are attractive for large scale systems because only one process needs to take an active part in the communication and because they can be implemented efficiently on high-performance communication systems.

In this paper, we focus on fault tolerance for applications that communicate using one-sided communications.

Fault-tolerance in parallel applications can be achieve at *system-level* or at *application-level*. Application-level fault tolerance can be integrated directly in the algorithm, relying on algebraic or algorithmic properties of the computations [11], [12], [13] or storing the data in the memory of other processes to be able to recover it upon failures [14]. It must be taken into account by the programmer.

System-level fault tolerance, on the other hand, is transparent to the programmer. Failure detection and the behavior upon failures is defined in the distributed execution environment and parallel applications do not need to be modified to be able to survive failures. Usually, this approach relies on checkpoint/restart mechanisms, where the state of each individual process is stored and recovered when required [15], [16], [17]. However, storing the state of the processes is not enough to maintain the consistency of the parallel application. Inter-process communications require specific attention in order to avoid creation of *orphan processes* after a rollback, *i.e.* processes that wait for a communication that will never happen.

The Chandy-Lamport algorithm relies on the notion of consistent cut in order to take a distributed snapshot of the whole application and be able to rollback on a consistent state [18]. We will see with a counter-example that former approaches to coordinate the processes and take this snapshot cannot be applied in a one-sided communication model.

The contributions of this paper are: a) we present a model for distributed systems using one-sided communication and discussing which hypothesis are made by current communication routines provided by the main parallel programming interfaces; b) in this model, we show why an implementation of the Chandy-Lamport algorithm cannot be transposed directly from two-sided communication models to one-sided communication model; c) we give three solutions to adapt the Chandy-Lamport algorithm and we compare them.

After an overview of some previous works on rollback recovery for fault tolerance in parallel, distributed applications in section II, we present the communication model in section III. In section IV, we present algorithms for coordinated checkpointing in this model, starting with a counter-example that shows why approaches used with two-sided communications do not work in this mode, then we present three possible algorithms and we compare them. Last, concluding remarks and perspectives come in section V.

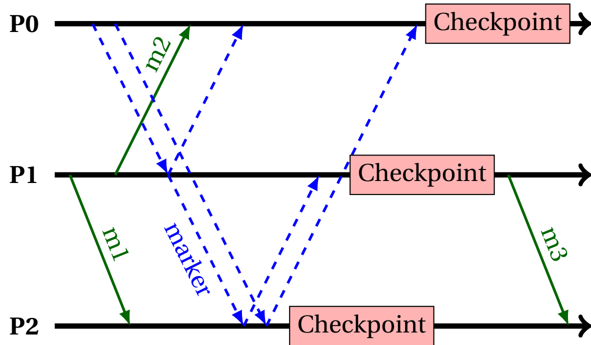

Figure 1. Checkpoint wave. Markers are represented with dashed lines

II. Related works

The goal of rollback-recovery protocols for fault tolerance in distributed applications is to maintain the consistency of the application, notwithstanding process failures and rollbacks. A survey of these protocols can be found in [19], [20]. Roughly speaking, these protocols fall into two categories: coordinated and non-coordinated checkpointing.

Non-coordinated checkpointing allows processes to take checkpoints independently from each other. However, communications need to be taken care of in order to recover a consistent state without forcing series of rollbacks, called *domino effect*. Message-logging protocols rely on an assumption: the execution of the processes of a distributed application is assumed to be *piecewise-deterministic*: each process execution is made of a sequence of deterministic segments interrupted by non-deterministic events (*e.g.* communications) [21]. Hence, messages sent between processes can be *logged and replayed* after a rollback [22], [23], [24], [25], [26], in the same causal order [27]. Which events are non-deterministic can be refined in order to reduce the number of events to log [28]. Another possibility to avoid domino effect consists in forcing checkpoints when inter-process dependencies are such that a failure would trigger a domino effect. This is called *communication-induced checkpointing* [29], [30], [31], [32].

Coordinated checkpointing was introduced in [18] and relies on the notion of *distributed snapshot*: a global snapshot, made of the snapshots of all the individual processes, is taken and stored. This global snapshot must represent a *consistent state*, *i.e.* not to be "crossed" by communications (*e.g.* a message sent by a process before it checkpoints its state and received by another process after it checkpoints its state, or this message does not introduce state change). This algorithm relies on the circulation of a marker that initiates a *checkpoint wave* at the end of which each process checkpoints its state (see Fig. 1). During the checkpoint wave, inter-process communications can be logged [33] or blocked until the end of the wave [34].

All the aforementioned literature considers a distributed system as a set of processes that communicate using two-sided communications, involving communication primitives such as *send()* and *recv()* that must be called on the source and destination processes and match with one another. However, modern high performance hardware implements another type of communication model: *Remote Direct Memory Access* (RDMA), involving one-sided communication primitives such as *put()* and *get()* [35], [36] (see Fig. 2). It is the communication model implemented in the high-performance programming interface OpenSHMEM [37], [10], [38].

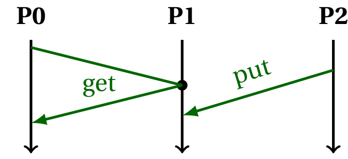

Figure 2. One-sided communications

There has been a need for fault-tolerance in OpenSHMEM [39], [40]. A user-level approach was presented in [41] and [42]. Both use a *shadow memory area* to store checkpoints in another process's memory, the latter taking explicit checkpoints and the former using redundant computations in order to maintain an up-to-date redundant copy in the shadow memory.

III. Model for distributed systems

A. Communication model

In this section, we define a model for one-sided communications. In this model, each process maps two distinct areas of memory: a *private* memory and a *public* memory. The private memory can be accessed from this process only. The public address space is made of the set of all the public memories of the processes (the *Global Address Space*). Processes can copy data from/to their private memory and the public address space, regardless of data locality.

Public memory can be accessed by any process of the application, in *concurrent* read and write mode. In particular, no distinction is made between accesses to public memory from a remote process and from the process that actually maps this address space.

In addition, since NICs (Network Interface Controllers) are in charge with memory management in the public memory space, they may provide *locks* on memory areas. These locks guarantee exclusive access on a memory area: when a lock is taken by a process, other processes must wait for the release of this lock before they can access the data.

Processes access areas of public memory mapped by other processes using point-to-point communications. They use *one-sided communications*: the process that initiates the communication can access remote data without any notification on the other process's side. Hence, a process A is not aware of the fact that another process B has accessed (*i.e.*, read or written) its memory. Accessing data in

another process's memory is called *Remote Direct Memory Access* (RDMA). It can be performed with no implication from the remote process's operating system by specific NICs, such as InfiniBand and Myrinet technologies. It must be noted that the operating system is not aware of the modifications in its local shared memory. The SHMEM library [10], developed by Cray, also implements one-sided operations on top of *shared* memory. As a consequence, the model and algorithms presented in this paper can easily be extended to shared memory systems.

RDMA provides two communication primitives: *put()* and *get()*. These two operations are represented in Fig. 2. They are both generally presented as *atomic* on the paper but, in practice, some implementation consider them as non-blocking and/or asynchronous and non-atomic (see section III-B).

B. Implementation in parallel programming libraries

MPI 2.0 and MPI 3.0 introduced one-sided communications based on these *put()* and *get()* primitives in remote windows. Communication routines are *asynchronous* and *non-blocking*, therefore, when they return, the data transfer may not be completed. Hence, explicit synchronization must be used to make sure the data has been transferred. The data remote processes can access is located in a *window*, which is an area of public memory; what remote processes see in this window can be identical (unified mode) or may be different (separated mode) from what the local process sees.

OpenSHMEM and many PGAS languages such as Unified Parallel C define one-sided communications on explicitly shared data and also based on the aforementioned *put()* and *get()* primitives. They support both blocking and non-blocking communication routines (introduced in OpenSHMEM 1.3 and UPC 1.3).

In [8], MPI one-sided communication routines are implemented directly on top of InfiniBand *read* and *write* operations. Non-blocking routines use the offloading features of some NICs (such as InfiniBand) so the communication is handled by the NIC itself in the background.

The number of communications that are actually performed by the *put()* and *get()* operations depends on the available hardware. InfiniBand provides *read* and *write* operations, but their implementation depends on the queue pair used.

Using a Reliable Connected (RC) queue pair, the requester considers a message operation complete once there is an ack from the responder (*i.e.* target) side that the message was read/written to its memory. The requester considers a message operation complete once the message was read/written to its (local) memory. Using an Unreliable Connected (UC) queue pair, the operation is considered as complete once all the data has been send (write) or once a complete message in correct sequence has been received by the received and written to its (local) memory (read).

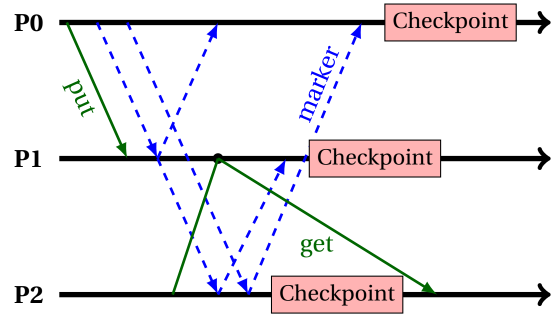

Figure 3. Counter-example: the data response crosses the cut.

The Unreliable Datagram (UD) queue pair supports Send operations only [43], [44].

With either RC and UC queue pairs, *read* operations are performed as follows: after a connection has been established, the requesting process (*i.e.* source process) sends a *read request* to the responder (*i.e.* target process) that reads its local memory and returns it.

Moreover, when the data to be transferred is larger than a local limit (the Path MTU, or PMTU), it is segmented in several response packets.

Therefore, in practice, one-sided communications cannot be assumed to be atomic, unless specified explicitly (*e.g.* for MPI, `MPI_Fetch_and_op`, `MPI_Compare_and_swap`...). Moreover, communication links are *full-duplex*, *i.e.* they support simultaneous data transfers in both directions.

IV. Algorithms

The original Chandy-Lamport algorithm [18] and its blocking [34], [45] and non-blocking [33] implementations take advantage of the FIFO property of the communication channels. In [18], a marker circulates between the processes to initiate the checkpoint wave, and the processes *flush* their communication channels during the checkpoint wave. The FIFO property guarantees that no message passes the marker and crosses the checkpoint wave. A short description of how this algorithm works and some options left to the implementation are given in section II.

However, although this is true with two-sided communications, the round-trip nature of the *get()* primitive on one-sided communications makes it less trivial with one-sided communications. For instance in Fig. 3, the read request of the *get()* is issued before P_2 receives the checkpoint marker and therefore, before P_2 enters the checkpoint wave, but the result of the *get()* is received by P_2 after it has entered the checkpoint wave. The FIFO property of the communication channel is still valid, but the round-trip pattern of the *get()* on duplex communication channels make it possible for a communication to cross the checkpoint line and therefore, breaks the consistency of the cut.

Now let us introduce some solutions to this problem.

A. Solution 1: Message delay during the communication wave

In this algorithm, when a process receive a marker, it switches to the state `checkpointing`, and once he has taken its checkpoint, it exits the checkpoint wave and switches to state `normal`. When a process issues a *get()* to a target process which is in `checkpointing` state, the target process *delays* the operation on its memory.

Property 1: No data movement crosses the cut made by this algorithm. However, some communications can overlap the checkpoint wave (the *read request* of a *get()*).

Proof: Proving this property is trivial for two-sided communications (see [18]) and *put()* communications (by FIFO property of the communication channels). Regarding the *get()* operations, the delay on delivery of the read request ensures that no message arrives on process t while it is in `checkpointing` state. Therefore, by FIFO property of the communication channel between s and t, the second step of the *get()* (data response) happens either before or after the checkpoint.

Note that a *get()* started before the source process gets into `checkpointing` state can return after the checkpoint is taken: a *get()* can overlap the checkpoint wave. ■

a) Practical concerns: Since, in RDMA communications, the target process does not take an explicit active part of the communication, this delay must be implemented at the NIC-level. When the NIC receives a *read* request, it checks the local state of the process and if the process is in `checkpointing` state, the NIC delivers the response only once the process returns to `normal` state.

B. Solution 2: Peek-and-get

It is possible to *delay the beginning* of a *get()* operation if the remote process is in a checkpoint wave. In this algorithm, when a *get()* is issued, the source process sends an atomic *read request* to read the state of the target, called here a *peek* (see Fig. 4). While the target is in `checkpointing` state, it keeps polling on this state. If the target is in `normal` state, the *get()* can be issued; a regular *read request* is issued and the response is sent by the target process.

Property 2: Using the peek-and-get algorithm, no communication crosses the cut made by this algorithm and no communication overlaps the cut made by this algorithm.

Proof: Proving this property is trivial for two-sided communications (see [18]) and *put()* communications (by FIFO property of the communication channels). Regarding the *get()* operations, let us examine the three possible scenarios between the checkpoint wave and the communications.

If a *read request* arrives on process t while it is in `normal` state, the data is received before the marker and therefore, before the source process enters the checkpoint wave.

If a *read request* arrives on process t while it is in `checkpointing` state, the *get()* operation will be delayed until the checkpoint wave is over. In other words, this operation will happen completely after the cut and, if the system rolls back on this global snapshot, the whole operation will be performed again.

If the atomic peek is performed while the target process is in `normal` state, then a marker is received from another process before the *get()* is received by the target process, the target process switches state between the *peek* and the actual *get()*. In this case, the target process sends a marker to the source process before it sends the result of the *get()*, since the communication channels have the FIFO property, reception of the marker will occur before the reception of the return of the *get()* by the target. Therefore, the source process switches to `checkpointing` state just after the end of the *get()* operation Therefore, all the *get()* operation is performed before the cut and, if the system rolls back on this global snapshot, the whole operation will already be done and will not have to be performed again. ■

Figure 4. Peek-and-get delaying a *get()* operation until the target switches back to `normal` mode.

Figure 5. Double barrier

C. Solution 3: Double barrier

To cope with the two-way message pattern of the *get()* operation, we may introduce a second all-to-all communication over the network to be sure that all communication channels are flushed (see Fig. 5).

Property 3: If a checkpoint wave is made of two successive all-to-all marker circulations and no application communication is initiated after the beginning of the circulation of the first marker, no communication crosses the cut made at the end of the second marker circulation and no communication overlaps this cut.

Proof: Trivial, since the communication channels are FIFO, communications with a back-and-forth pattern (such as *get()*) starting before a process enters the checkpoint wave finish before the double circulation of the marker, since it involves two causally ordered communications in each direction between each pair of processes. ■

One can argue that only pending *get()* operations are of concern in this second wave. If we keep a list of all these pending *get()* operations, we can flush only the channels those concerned by these *get()* operations, *i.e.* the channels between pairs of processes concerned by each *get()*. In the worst case, it is not better than a full flooding of the communication network (actually, it is equivalent), but in practice, it may introduce an interesting reduction of the added cost to the first wave by flushing only some channels.

Practically speaking, if a marker is received by a *source* process during a *get()* operation with a *target* process (*i.e.*, if the *read request* and the marker cross each other on the communication channel), the source process sends a specific marker to the target process and waits until it receives a second marker from the target process. This marker is identified as particular by the *target* process that determines it is a type of marker that expects an acknowledgement. We call this algorithm a *partial double barrier*.

a) Remark: During this force-flushing procedure of the partial double barrier, only three markers are exchanged between the two processes *source* and *target*.

Proof: The *source* process cannot initiate any communication during the checkpoint wave. Therefore, when it started the *get()* operation, the *source* process was in *normal* mode: it had sent no marker since the previous checkpoint wave. ■

Property 4: Using the partial double barrier algorithm, no communication crosses the cut made by this algorithm and no communication overlaps the cut made by this algorithm.

Proof: If a marker is received by a *source* process between the two steps of a *get()* with the same process, a specific marker is sent to the *target* process that will answer it by another marker. Therefore, since the communication channels have the FIFO property and since the *source* process waits until completion of the *get()* communication before it switches to `checkpointing` state and sends its markers, all the *get()* operation is performed before the cut, so it neither cross not overlaps the cut. ■

D. Comparison

a) Performance: First let us remark that flushing all communication channels costs *at least* to send a message accross all the channels between P_i and P_j with $i \neq j$. Let us denote n the number of processors. The number of such channels is $n(n-1)$ since we have to distinguish channel P_i—P_j from P_j—Pi. So obtaining a consistant snapshot is at least in $O(n^2)$.

The algorithm of [18] use such a flooding and is optimal in two-sided communication models since it costs $n(n-1)$ messages.

TABLE I. CONSISTENCY PROPERTIES OF THE CUT MADE BY THE ALGORITHMS REGARDING ONE-SIDED COMMUNICATIONS.

	Overlap	Cross
Vanilla	no	no
Delay	no	yes
Peek-and-get	yes	yes
Double barrier	yes	yes

Solution 1 introduces no additionnal message and is therefore of great interest *when available*. If the *get()* operation is non-blocking, it may be rather complex to implement this: either by specific hardware or by flashing new procedures in a programmable hardware.

Solution 2 is clearly heavy in terms of additional messages exchanged. Moreover, the checkpointing may takes a long time (at least proportionnal to n) and the atomicity of the peek-and-get is mandatory, again this may be rather complex to implement.

Solution 3 with a full second flooding phase adds $n(n-1)$ messages. So it is still in $O(n^2)$, and then is still optimal. However, as we already stated it, it can be practically improved. If g denotes the number of pending *get()* operations that the first wave encounters, the improved procedure adds only $2g$ messages. The worst case is obtained when each processor has started a non-blocking *get()* toward all other processors. Note that even if P_i starts more than one *get()* toward P_j, it is enough to empty the channel P_i—P_j and the way back, that is to say a message for all channels. This gives a total of $n(n-1)$ additional messages. So even in this worst case, this improved procedure is not worse than the normal flooding.

b) Consistency: The properties of the algorithms presented here and the vanilla Chandy & Lamport algorithm are summarized in table I. We have seen that with the *delay* algorithm, *get()* communications can overlap the checkpoint wave but they cannot cross the cut (section IV-A). Practically speaking, it means that after a global rollback, the first step of a *get()* communication (the *read* request) can be considered as already sent by the source process, which is waiting for the data to be transferred after the checkpoint wave. This is not a problem if the state of the NICs is included in the local snapshots. In this case, the source process rolls back in a state that waits for the data, while the target process rolls back in a state in which the NIC engine is about to deliver the *read* request (stored in the checkpoint and therefore, delivered after the rollback). However, if the state of the NICs is not stored in the checkpoints, cuts that are overlapped by *get()* communications cannot restore a consistent state.

V. CONCLUSION

In this paper, we have identified a counter-example that exhibits the fact that the Chandy-Lamport algorithm for

coordinated checkpointing of a distributed system by determination of a consistent cut cannot be applied directly on a system that communicates using Remote Direct Memory Access or one-sided communications.

We have presented a model for one-sided communications in such distributed systems, and discussed its relevance with respect to the implementation of one-sided primitives in current *de facto* parallel programming standards.

The whole challenge of taking a snapshot of a distributed system is related to the inter-process communication channels. The Chandy-Lamport algorithm relies on a coordination between the processes that flushes these communication channels. In this model for one-sided communications, we have presented three solutions to flush the communication channels during the coordination of the process in the Chandy-Lamport algorithm.

Last, we have discussed the properties of these algorithms with respect to the hypothesis provided by the possible implementations of the model, and the complexity of these algorithms, in particular their overhead compared to the original Chandy-Lamport algorithm for two-sided communications.

This model suits well the current parallel programming systems: it can be used to examine other algorithms, and to reason on distributed algorithms that target current high-performance systems. Besides, another perspective open by this work is the implementation and practical evaluation of fault-tolerant execution systems.

References

[1] "Top500," http://www.top500.org.

[2] "Reliability challenges in large systems," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 293 – 302, 2006.

[3] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of Blue Waters," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 610–621.

[4] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[5] J. Dongarra *et al.*, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, 2011.

[6] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, 2009.

[7] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *International Conference on High Performance Computing for Computational Science*. Springer, 2010, pp. 1–25.

[8] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, W. Gropp, and R. Thakur, "High performance mpi-2 one-sided communication over infiniband," in *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*. IEEE, 2004, pp. 531–538.

[9] C. Maynard, "Comparing one-sided communication with mpi, upc and shmem," *Proceedings of the Cray User Group (CUG)*, vol. 2012, 2012.

[10] H. P. C. T. group at the University of Houston and O. R. N. L. Extreme Scale Systems Center. (2012, Jan.) OpenSHMEM application programming interface, version 1.0 final. http://www.openshmem.org.

[11] C. Coti, "Scalable, robust, fault-tolerant parallel qr factorization," in *Distributed Computing and Applications to Business, Engineering and Science (DCABES), 2016 15th International Symposium on*, S. Khaddaj, Ed. IEEE, 2016.

[12] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *J. Parallel Distrib. Comput.*, vol. 69, no. 4, pp. 410–416, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2008.12.002

[13] 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 *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2005, pp. 213–223.

[14] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 972–986, Oct. 1998. [Online]. Available: http://dx.doi.org/10.1109/71.730527

[15] M. J. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the condor distributed processing system," University of Wisconsin-Madison, Tech. Rep. 1346, 1997.

[16] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," in *USENIX Winter*, 1995, pp. 213–224.

[17] E. R. Jason Duell, Paul Hargrove, "The design and implementation of berkeley lab's linux checkpoint/restart," Berkeley Lab, Tech. Rep. publication LBNL-54941, 2003.

[18] 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.

[19] E. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message passing systems," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep. CMU-CS-96-181, October 1996.

[20] ——, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375 – 408, september 2002.

[21] R. E. Strom and S. A. Yemini, "Optimistic recovery in distributed systems," in *Transactions on Computer Systems*, vol. 3(3). ACM, August 1985, pp. 204–226.

[22] S. Rao, L. Alvisi, and H. M. Vin, "The cost of recovery in message logging protocols," in *17th Symposium on Reliable Distributed Systems (SRDS)*. IEEE CS Press, October 1998, pp. 10–18.

[23] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Trans. Software Eng*, vol. 24, no. 2, pp. 149–159, 1998.

[24] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *Journal of algorithms*, vol. 11(3), pp. 462–491, March 1990.

[25] ——, "Sender-based message logging," in *The 17th annual international symposium on fault-tolerant computing (FTCS'87)*. IEEE CS Press, 1987.

[26] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *High Performance Networking and Computing (SC2003)*, Phoenix USA. IEEE/ACM, November 2003.

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

[28] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic MPI applications," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 989–1000.

[29] J.-M. Hélary, "Observing global states of asynchronous distributed applications," in *Proceedings of the 3rd International Workshop on Distributed Algorithms*. London, UK: Springer-Verlag, 1989, pp. 124–135.

[30] J. Tsai, S.-Y. Kuo, and Y.-M. Wang, "Theoretical analysis for communication-induced checkpointing protocols with rollback-dependency trackability," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 963–971, 1998.

[31] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel, "An analysis of communication induced checkpointing," in *29th Symposium on Fault-Tolerant Computing (FTCS'99)*. IEEE CS Press, june 1999.

[32] J.-M. Hélary, 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.

[33] 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.

[34] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI," in *Proceedings of the International Conference for High Performance Networking Computing, Networking, Storage and Analysis (SC|06)*, ACM/IEEE, Ed., Tampa, USA, November 2006, p. electronic.

[35] F. Butelle and C. Coti, "A model for coherent distributed memory for race condition detection," in *proceedings of the 13th Workshop on Advances in Parallel and Distributed Computational Models (APDCM'11)*, Anchorage, Ak, May 2011, pp. 579–585.

[36] ——, "Data coherency in distributed shared memory," *The International Journal of Networking and Computing*, vol. 2, no. 1, pp. 117–130, 2012.

[37] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010, p. 2.

[38] C. Coti, "POSH: Paris OpenSHMEM: A High-Performance OpenSHMEM Implementation for Shared Memory Systems," vol. 29, no. 0, 2014, pp. 2422 – 2431, 2014 International Conference on Computational Science.

[39] P. Hao, P. Shamis, M. G. Venkata, S. Pophale, A. Welch, S. Poole, and B. Chapman, "Fault tolerance for openshmem," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14, 2014, pp. 23:1–23:3.

[40] T. Curtis and B. Chapman, "Check-pointing approach for fault tolerance in openshmem," in *OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies: Second Workshop, OpenSHMEM 2015, Annapolis, MD, USA, August 4-6, 2015. Revised Selected Papers*, vol. 9397. Springer, 2015, p. 36.

[41] N. Ali, S. Krishnamoorthy, N. Govind, and B. Palmer, "A redundant communication approach to scalable fault tolerance in PGAS programming models," in *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*. IEEE, 2011, pp. 24–31.

[42] P. Hao, S. Pophale, P. Shamis, T. Curtis, and B. Chapman, "Checkpointing approach for fault tolerance in openshmem," in *Workshop on OpenSHMEM and Related Technologies*. Springer, 2014, pp. 36–52.

[43] D. Barak, "Which queue pair type to use," http://www.rdmamojo.com/2013/06/01/which-queue-pair-type-to-use/.

[44] T. Shanley, *Infiniband Network Architecture*. Addison-Wesley Professional, 2003.

[45] 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.