A Resilient Hierarchical Checkpointing Algorithm for Distributed Systems Running on Cluster Federation

Houssem Mansouri[1](✉) and Al-Sakib Khan Pathan[2]

[1] Laboratory of Networks and Distributed Systems, Computer Science Department, Faculty of Sciences, Ferhat Abbas Setif University 1, Sétif, Algeria
mansouri_houssem@univ-setif.dz

[2] Department of Computer Science and Engineering, Southeast University, Dhaka, Bangladesh
spathan@ieee.org

Abstract. In distributed systems, checkpointing method can be used to ensure fault-tolerance, which in turn could help the system's security and stability. In simple terms, checkpointing means saving status information of a system. In this work, a novel hierarchical checkpointing algorithm for distributed systems is proposed, which runs on cluster federation. This algorithm is based on two well-known techniques in the literature and it ensures that a locally consistent state is always maintained in each cluster with a global optimistic logging technique between clusters. The proposed algorithm synchronizes the intra- and inter-cluster checkpointing process in such way that each checkpoint taken locally in any cluster is a segment of the consistent global state. Compared to other works, our scheme has low message cost as it makes sure that only few processes in each cluster record checkpoints for any execution operation.

Keywords: Checkpointing · Cluster-based · Consistent global state · Distributed systems · Fault tolerance · Non-blocking · Optimistic logging · Recovery

1 Introduction

Today, cluster architectures are very widely spread in the research arena and in the industries. However, the use of computer cluster raises a number of problems related to several factors imposed by the nature of the distributed computing environment. In this kind of environment, resources can be volatile because the nodes can get disconnected when the connection is lost or hardware failure occurs, or voluntarily during proprietary use. In addition, the cluster may be exploited by a large number of users with very large distributed applications with long runtime. Thus, the risks of occurrence of faults become very high; these faults would cause failures that prevent the correct execution of the distributed applications – eventually, threatening the stability and security. Hence, to deploy computing applications on a large number of nodes, it is a necessity to have effective fault-tolerance mechanism.

In distributed systems, fault-tolerance can be ensured by using checkpointing techniques. A checkpointing technique basically maintains the records of the system on

S. M. Thampi et al. (Eds.): SSCC 2019, CCIS 1208, pp. 99–110, 2020.
https://doi.org/10.1007/978-981-15-4825-3_8

some stable storage when the system is running without any fault (i.e., fault-free operation state). In case of any system failure, it could then restart from a previously recorded consistent global checkpoint state (i.e., according to the preserved record). Though fault-tolerance and security are two different terms, they are often interrelated. If a system is not fault-tolerant, unstable state of it or not functioning in the expected way (after a failure) could open several security loopholes as well that could be exploited by rogue entities.

Usually, in distributed environments, application processes communicate by passing messages among themselves. In this kind of computing environment, some kind of casual dependency is induced by a message between a transmitter process and a receiver process. The global dependency of the distributed application is defined by the transitive dependencies of the messages. Hence, only restarting any faulty process is not enough after a fault occurrence. The employed checkpointing algorithm must also ensure that all the related processes are coherent after the recovery operation is executed and it must maintain the dependencies between the applications. Hence, all these are also very important for the system's overall level of security.

In the existing literature, there are various types of checkpointing recovery algorithms. In general, all of the algorithms could be classified into two major categories:

- *Checkpoint-based techniques*: Each process periodically saves its state into a stable storage disk. After occurrence of fault, all processes roll back to a coherent global state. In this way, the strategy limits the amount of lost computation.
- *Message logging techniques*: Messages are saved so that they can be replayed in the same order in case of fault; only the faulty processes roll back to a coherent local state.

Main contribution of this paper is proposing a hierarchical checkpointing algorithm for distributed systems running on cluster federation. Our proposed algorithm results from a thorough comparative study of different algorithms/techniques proposed in the literature. Essentially, it is a combination of a *non-blocking* type checkpointing technique and the pessimistic message logging technique [1]. Here, the cluster is the main component in which the non-blocking checkpointing algorithm will be executed in a simultaneous way and the inter-cluster messages will be saved using a pessimistic logging technique based on the receiver process.

After this introductory section, in Sect. 2, our system model is presented. Section 3 discusses the related works. Section 4 proposes our checkpointing strategy. Performance evaluation along with comparisons is presented in Sect. 5. Section 6 presents the simulation results. The paper is concluded with Sect. 7 in which we also mention the possible future research directions.

2 System Model

Cluster federation (Fig. 1) is a union of n number of processes distributed on m number of clusters where k processes are contained by each cluster [2–4]. A SAN (System Area Network) connects the processes within a cluster. A Local Area Network (LAN) or

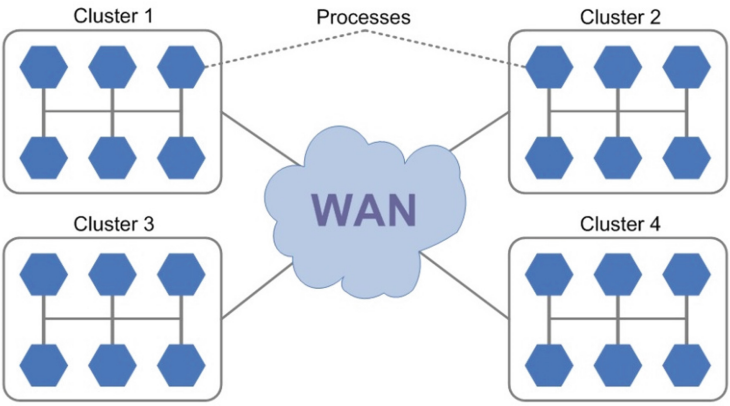


Fig. 1. Clusters in a cluster federation.

Wide Area Network (WAN) connects the clusters in the federation. In such environment of cluster computing, a running distributed application is divided into several modules which communicate with each other. The modules run on different process from different clusters.

Let us consider that there are *n* number of concurrent processes, *P1, P2, P3,..., Pn* which are running on *n* clusters, at a rate *k* processes by cluster, *P1, P2, P3,..., Pk.* In this setting, we assume a *Fail-stop* mode. This means that when there is a process failure, it would immediately stop its execution. In this way, it would refrain from doing any malicious operation. Otherwise, it could threaten the security of the system. Reliable inter-cluster and intra-cluster message deliveries are also assumed in this setting. This means that during normal computation, there would not be any loss or alteration of message. In addition, there is no shared memory, no common or synchronized clock, or any central coordinator. Message passing is the only mode of communication between any pair of processes (inter- or intra-cluster). It is possible that the checkpointing process can be initiated by any process. There are finite but arbitrary delays when the message exchanges take place [5–8].

3 Related Work

In this section, some related works have been discussed. These are the representative previous works that helped us come up with our scheme.

There are three prominent schemes for checkpointing and rollback recovery in the existing literature. They are (i) *central file server checkpointing*, (ii) *checkpoint mirroring*, and (iii) *skewed checkpointing*. Using these schemes, the researchers in [9] develop a stochastic model to evaluate the expected total recovery overhead for a cluster computing system. Their work also presents a comparative study considering expected total recovery overhead. Generally, in a cluster system, there could be either single-process

failure or multi-process failure. Hence, it is often difficult to assess the closed form of expected total recovery overhead. By performing various quantitative comparisons, the authors show that the skewed checkpointing is superior to other alternative schemes.

A recovery approach for handling both inter-cluster orphan and lost messages is proposed in [10]. This algorithm is executed simultaneously by all the clusters in a cluster federation for determining the recovery line. In this way, it is ensured that any inter-cluster orphan message may not exist between any pair of cluster level checkpoints belonging to the recovery line. A sender-based message logging technique is applied to handle inter-cluster lost messages for ensuring the correctness of computation.

A single-phase non-blocking checkpointing scheme is presented in [8]. This scheme ensures that after a failure-recovery, all processes in a system in different clusters can restart from their respective most recent checkpoints. In this way, *domino-effect* could be avoided. This basically implies that the most recent checkpoints can form a consistent recovery line for the cluster federation almost in all cases. The work takes advantage of message logging which enables the initiator process in each cluster to log minimum number of messages. One good feature of this approach is that it does not depend on the number of processes that may concurrently fail in a given setting.

A hierarchical checkpointing protocol is presented in [3], which is suitable for code coupling applications. The degree of interdependence between software modules is called coupling (generally). This is basically a measure of how closely two modules are connected with each other or the strength of their relationships. Code coupling applications show good level of code relationships. In this work, the authors' approach relies on a hybrid method of combining coordinated checkpointing within clusters. In between the clusters, some communication-induced checkpointing works. Though the work is fine, the algorithm needs some enhancements by adding some transitivity in the dependency tracking mechanism. In fact, as the claim is that the algorithm could tolerate multiple faults in a cluster, this also means that the redundancy is more for implementation of stable storage.

A fast and efficient recovery algorithm is proposed in [11] for cluster computing environment. The key aspect of this algorithm is that it does not depend on the cluster federation's architecture. This algorithm can also be run simultaneously by all participating clusters when determining a federation-level recovery line. Also, when compared with the approach in [3], it shows that it reduces computational overhead significantly.

In [5], the authors propose a low-cost non-blocking checkpointing algorithm for cluster federation. The time interval between successive invocations of the algorithm is significant here as that ensures minimum number of lost or delayed messages. Major aspects of the scheme are: (a) minimum number of processes take the checkpoints, (b) the communications between clusters are kept at a minimum, (c) for speedy execution, a decentralized approach is used which ensures that each cluster would maintain its own data structures to store the checkpointing dependency information, and (d) bandwidth wastage is kept at a minimum. In [6], the proposed *non-blocking* checkpointing/recovery algorithm limits the effect of domino phenomenon by the time interval between successive invocations of the algorithm. Recovery in this approach is as simple as that in the synchronous approach presented in [5]. A key aspect of this approach is that it

employs some kind of responsibility taking strategy of the sender of a message to make it *non-orphan*.

In [7], another hybrid checkpointing algorithm is proposed which combines coordinated and communication-induced checkpointing methods. Based on the network and application communication pattern, this algorithm can be tuned accordingly. The authors have evaluated the algorithm via simulation studies and showed that the algorithm is suitable for applications that can be divided into several modules where many communications happen within a single module but communications in between the modules are relatively less.

4 Proposed Algorithm

4.1 Basic Idea

The basic idea of our new checkpointing algorithm is to combine the non-blocking checkpointing technique [12, 13] applied within clusters with the pessimistic message logging technique [14] applied between clusters. However, within the clusters, processes will be coordinated by the checkpointing process using non-blocking technique. The advantage of using this solution in intra-cluster setting is that during its execution, the logging protocol could run (at the same time) on the inter-cluster messages. These saved messages may not be part of the calculation for the system's global state, since they will be replayed only in case of failure.

For the adaptive algorithm to switch to this combination, the number of exchanged messages (by the processes) between clusters must reach to a sufficiently high threshold value. The experiments allowed the threshold to be set at five (5) messages per second (5 msg/s). Hence, if the frequency of inter-cluster communication messages exceeds this threshold, the logging of these messages degrades the performance of the distributed applications. At the same time, to save the process states, the non-blocking coordinated checkpoint algorithm will be used synchronously (and simultaneously) inside each cluster at a checkpointing frequency of 180 s.

The purpose of our proposed algorithm is to reduce the message cost during normal execution, and also to avoid a too-long recovery procedure that could slow down the operation of the distributed system in the event of a failure. For this, it is assumed at first that the running distributed application generates a few intra-cluster messages under a maximum frequency threshold.

4.2 Checkpointing Implementation

The combined checkpointing algorithm is presented in Subsect. 4.3 (Fig. 2 and Fig. 3). In this algorithm, each process has an identifier *Pi* and the identifier of its cluster *Cj*. At the start of the distributed application execution, it is the optimistic logging-based checkpointing algorithm that runs *intra-cluster* [1]. Here, all inter-cluster messages are saved by applying optimistic logging-based technique on the receiver process memory

in the *InpMsg* set. Similarly, the determinant of each message is recorded at the level of receiver memory in the *DetMsg* set. The determinant is composed of: the sending date, the receipt date, and the message sequence number. To save a new process checkpoint, the algorithm stores the following data on a stable storage: the state of the process, the state of the incoming channels, and the state of the outgoing channels. At the same time, the non-blocking coordinated checkpoint-based algorithm [15, 16] runs synchronously (and simultaneously) inside each cluster. In our scheme, It is launched in every one hundred and eighty seconds (180 s), in which, *Pi* (which is basically the initiator process) saves a temporary checkpoint. Then a checkpoint request is sent by *Pi* to its directly dependent process, $Pj \in DepProcsi$ (*Pj* is the process that sends a computing message to *Pi* after taking its last snapshot). The *Ckpti* index and a value t = *1/Card(DepProci)* (*Card* refers to the cardinal function) are piggybacked by this request. This implies that element number in *DepProci* set is provided by *Card(DepProci)*.

If a checkpoint request is received by process *Pj* from another process *Px* during the running period of the algorithm, an answer is sent by *Pj*, then it piggybacks *t*'s value to the initiator process if the set *DepProcj* is empty. If not, it sends a checkpoint request piggybacking a new value, *t = t/Card(DepProcj)* to its directly dependent process (which is in fact, indirectly dependent on *Pi*), and so on. Whenever *Pi* (i.e., the initiator process) receives an answer, it collects the value *t* in *Termi*. If *Termi* is equal to 1, it sends a validation request to all other processes running in the cluster. If a process receives that validation request, it is required to store its temporary checkpoint as a permanent/stable one. Then, it would reset its data structure.

4.3 Data Structure and Pseudo Code

Every process *Pi* in different clusters has the following data structure:

- P_i*: process id (i ∈ [1…n]).*
- C_j*: cluster id (j ∈ [1…m]).*
- $Ckpt_i$*: index of the last checkpoint saved.*
- $DepProc_i$*: set of dependant processes in the same cluster.*
- $Term_i$*: algorithm termination detection.*
- $TempCkpt_i$*: last temporary snapshot.*
- $PermCkpt_i$*: last permanent snapshot.*
- $InpMsg_i$*: set of messages sent by the process.*
- $DetMsg_i$*: set of determinants of messages receipt by the process.*

Every message determinant has the following data structure:

- *SeqNbr: sequence number.*
- *SntDate: sent date.*
- *RcpDate: receipt date.*

Algorithm Part 1: *Checkpointing Process*

Part executed always by every process on each cluster

Upon receiving a message Msg:

if ($C_{Receiver} \neq C_{Sender}$)
Save *Msg to* $InpMsg_{Receiver}$;
Save *Determinant to* $DetMsg_{Receiver}$;
else
if ($Ckpt_{Receiver} < Ckpt_{Sender}$)
Save ($TempCkpt_i$);
$Ckpt_{Receiver} \leftarrow Ckpt_{Sender}$;
endif
endif

Part executed simultaneously on each cluster every 180 s

Part executed by the initiator process Pi

Save ($TempCkpt_i$);
$Ckpt_i := Ckpt_i + 1$;
for All ($P_x \in DepProc_i$)
Send *Checkpoint Request* ($Ckpt_i$, t);
endfor
Upon receiving : Answer Request (t)
$Term_i := Term_i + t$;
if ($Term_i = 1$)
for All ($P_x / x \in [1..k]$)
Send *Conformation Request ();*
endfor
endif

Part executed by every process Pj in the cluster

Upon receiving : Checkpoint Request ($Ckpt_x$, t)
if ($Ckpt_j \leq Ckpt_x$)
Save ($TempCkpt_j$);
$Ckpt_j := Ckpt_j + 1$;
if ($DepProc_j = \varnothing$)
Send *Answer Request (t);*
else
for All ($P_x \in DepProc_j$)
Send *Checkpoint Request* ($Ckpt_j$, t);
endfor
endif
endif
Upon receiving : Conformation Request ()
$PermCkpt_j \leftarrow TempCkpt_j$;
$InpMsg_j \leftarrow \varnothing$;
$DetMsg_j \leftarrow \varnothing$;
$Term_j \leftarrow 0$;

Fig. 2. Algorithm part 1.

Algorithm Part 2: *Recovery Process*

Part executed by the faulty Process

for All *(P_x / $x \in [1..n]$)*
Send *Recovery Request ();*
endfor

Part executed by every process Pj in the system

Upon receiving a Recovery Request ()

if *($C_{Receiver} = C_{Sender}$)*
Resume execution at the last recorded checkpoint $PermCkpt_j$;
else
Replay the reception event of $Msg \in InpMsg_j$ based on $DetMsg_j$;
endif

Fig. 3. Algorithm part 2.

4.4 Recovery Implementation

After resuming from fault, all the processes in the cluster containing the faulty process resume execution at the last checkpoint recorded during the last coordinated checkpointing process. At the same time, all processes in others clusters replay the reception event of all the messages received from the cluster containing the faulty process (after its last recorded checkpoint). Here, their reception orders are saved in the determinants recorded in *DetMsg* at the time of execution (without fault).

5 Comparison of the Performances of Various Algorithms

Table 1 shows the performances of various checkpointing algorithms for cluster federation. Nine significant evaluation criteria are used for this comparative study. It is clear that our algorithm shows some advantages compared to other existing alternatives.

The main aspects that make our proposed mechanism relatively more efficient than the other alternative mechanisms are:

(1) The basic idea of our algorithm is independent of all system architectures.
(2) As concurrent failures are taken care of, single failures are also well-tackled.
(3) For the recovery process, there is no *domino-effect*. The algorithm can guarantee the minimum re-computation in this process.
(4) Just the messages that a process has received after its most recent permanent checkpointing (from only intra-cluster processes) need to be logged by a process at its recent local checkpoint.
(5) The most recent local permanent checkpoint needs to be only saved by a process. Hence, the number of trips to stable storage during recovery per cluster is merely, k.
(6) In this approach, blocking of the execution of the distributed application is not needed.

Table 1. Comparative chart.

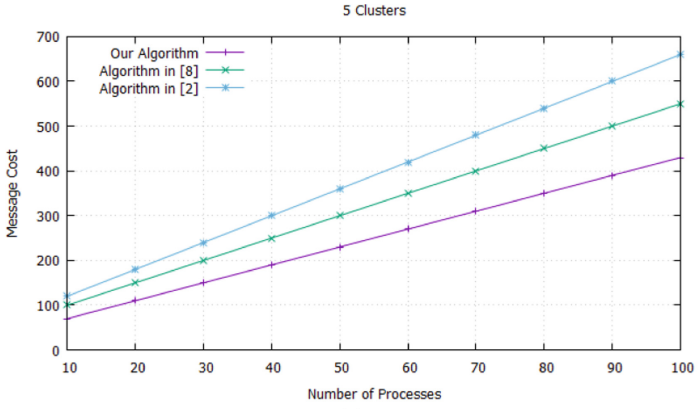
Criteria	[2]	[7]	[8]	[10]	[11]	[17]	Our algorithm
1. Dependent on architecture	*Yes*	*No*	*Yes*	*No*	*No*	*No*	*No*
2. Domino-effect free	*No*	*No*	*Yes*	*Yes*	*No*	*Yes*	*Yes*
3. Concurrent failures	*No*	*No*	*Yes*	*No*	*No*	*Yes*	*Yes*
4. Logging message inter-cluster	*No*	*No*	*Yes*	*Yes*	*No*	*Yes*	*Yes*
5. Blocking time	*0*	*0*	*0*	*0*	*0*	*0*	*0*
6. Simultaneous execution (by clusters)	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*
7. Number of checkpoints by process	>1	>1	1	1	>1	1	1
8. Stable storage related number of trips	$k + r$	$k + r$	k	k	$k + r$	k	k
9. Message complexity	$O(kn)$	$O(kn^2)$	$O(n)$	$O(kn)$	$O(kn)$	$O(n^2)$	$O(n)$

(7) The algorithm is run simultaneously by all the clusters in the system.
(8) Message complexity is pretty simple: $O(n)$.

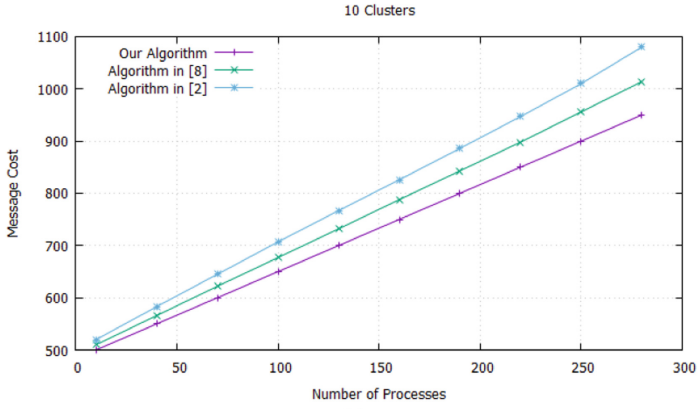
Table 1 explains that our proposed algorithm is of clear advantage over the algorithms proposed in [7, 10, 11], and [17]. We can also note that there are two algorithms which may be close to our proposed algorithm in terms of performance; especially, based on the message complexity criterion – these two are the algorithms in [2] and [8]. Hence, for these cases, we need to compare the message cost of the three algorithms against the number of processes and clusters in the system through different simulation scenarios. That is why we have also done some simulation studies to show the efficiency and clear superiority of our algorithm.

6 Simulation Results

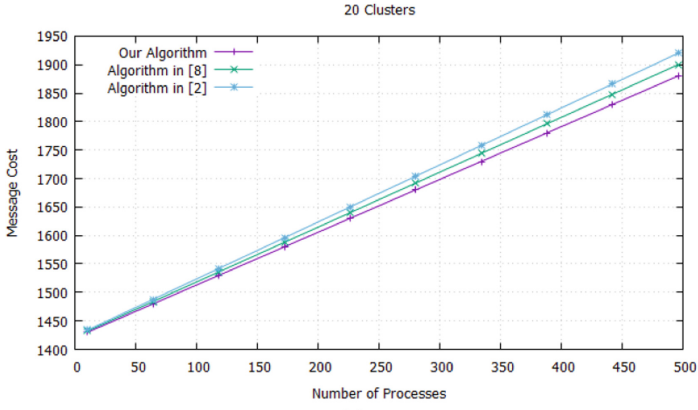
Message costs for [2, 8], and our algorithm to complete checkpointing processes considering the best case scenario are shown in Fig. 4(a), (b), and (c). We consider three clustering schemes: with 5, 10, and 20 clusters, against the number of processes in the system. *ChkSim* [18] simulator is used for these experiments. As the simulator is written in Java language, it is possible to run it on various platforms as long as the Java Virtual Machine (JVM) is available. We implemented the three checkpointing algorithms as Java classes.



(a)



(b)



(c)

Fig. 4. Message cost vs. number of processes in the system when: (a) m = 5 (b) m = 10 (c) m = 20.

In our experiments, we changed the number of clusters (5, 10, and 20) and varied the number of processes (from 10 to 20 * m) to see the corresponding changes of the message cost. Especially, we tried to compare the performance of our algorithm with the algorithms in [2] and [8].

It is evident that the other two algorithms in question show relatively higher message costs to determine a global consistent state as the numbers of processes and/or clusters increase in the system. It could also be noticed that the message cost of the algorithm in [2] is higher than that of the algorithm proposed in [8]. Hence, it is fair to state that our algorithm shows better efficiency in terms of message cost. In fact, when the numbers of processes and/or clusters decrease, our algorithm's message cost would be even lesser comparatively. Thus, this is relatively more suitable for saving a consistent global state compared to any other alternative approach.

7 Conclusions and Future Research Direction

Here, we proposed a hierarchical checkpointing algorithm combining an optimistic logging method applied to inter-cluster messages and a non-blocking checkpoint technique used in intra-cluster setting. This adaptive algorithm is of coordinated type and it avoids the blocking of the distributed application. It also minimizes the message cost to the strict necessary, which significantly decreases network traffic in favor of the distributed application. Overall, by this technique, the system's requirement of fault-tolerance and security could be supported. A performance evaluation and simulation comparison of our proposition with other reference algorithms proves the efficiency of our algorithm.

Further investigations can be done on this issue. For instance, taking various complex deployment scenarios, the algorithms could be tested for their performance. There is another direction for research; that the algorithm could be tested against multiple simultaneous executions.

References

1. Elnozahy, M., Alvisi, L., Wang, Y.-M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3), 375–408 (2002)
2. Cao, J., Chen, Y., Zhang, K., He, Y.: Checkpointing in hybrid distributed systems. In: Proceedings of the 7th IEEE International Symposium on Parallel Architectures, Algorithms and Network, pp. 136–141. IEEE Press, New York (2004)
3. Monnet, S., Morin, C., Badrinath, R.: A hierarchical checkpointing protocol for parallel applications in cluster federations. In: Proceedings of the 9th IEEE Workshop on Fault-Tolerant Parallel, Distributed and Network-Centric Systems, pp. 211–218. IEEE Press, New York (2004)
4. Sharma, K., Tomar, A., Kumar, M.: Clustering and recovery mechanism using checkpointing to improve the performance of recovery system. Int. J. Sci. Res. Dev. **4**(7), 827–832 (2016)
5. Kumar, M.: An efficient recovery mechanism with checkpointing approach for cluster federation. Int. J. Comput. Sci. Appl. **4**(6), 33–45 (2014)
6. Gupta, B., Rahimi, S., Ahmad, R.: A new roll-forward checkpointing/recovery mechanism for cluster federation. Int. J. Comput. Sci. Netw. Secur. **6**(11), 292–298 (2006)

7. Monnet, S., Morin, C., Badrinath, R.: Hybrid checkpointing for parallel applications in cluster federations. In: Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 773–782. IEEE Press, New York (2004)
8. Gupta, B., Rahimi, S.: Novel crash recovery approach for concurrent failures in cluster federation. In: Abdennadher, N., Petcu, D. (eds.) GPC 2009. LNCS, vol. 5529, pp. 434–445. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01671-4_39
9. Bessho, N., Dohi, T.: Comparing checkpoint and rollback recovery schemes in a cluster system. In: Xiang, Y., Stojmenovic, I., Apduhan, B.O., Wang, G., Nakano, K., Zomaya, A. (eds.) ICA3PP 2012. LNCS, vol. 7439, pp. 531–545. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33078-0_38
10. Gupta, B., Nikolaev, R., Chirra, R.: A recovery scheme for cluster federations using sender-based message logging. J. Comput. Inf. Technol. **19**(2), 127–139 (2011)
11. Gupta, B., Rahimi, S., Ahmad, R., Chirra, R.: A novel recovery approach for cluster federations. In: Cérin, C., Li, K.-C. (eds.) GPC 2007. LNCS, vol. 4459, pp. 519–530. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72360-8_44
12. Mansouri, H., Badache, N., Aliouat, M., Pathan, A.-S.K.: Adaptive fault tolerant checkpointing protocol for cluster based mobile ad hoc networks. Procedia Comput. Sci. **73**, 40–47 (2015)
13. Mansouri, H., Badache, N., Aliouat, M., Pathan, A.-S.K.: A new efficient checkpointing algorithm for distributed mobile computing. J. Control Eng. Appl. Inform. **17**(2), 43–54 (2015)
14. Mansouri, H., Pathan, A-S.K.: An efficient minimum-process non-intrusive snapshot protocol for vehicular ad hoc networks. In: Proceedings of the 13th ACS/IEEE International Conference on Computer Systems and Applications, pp 83–92. IEEE Press, New York (2016)
15. Mansouri, H., Pathan, A.-S.K.: Checkpointing distributed application running on mobile ad hoc networks. Int. J. High Perform. Comput. Netw. **11**(2), 95–107 (2018)
16. Mansouri, H., Pathan, A.-S.K.: Checkpointing distributed computing systems: an optimization approach. Int. J. High Perform. Comput. Netw. **15**(3–4), 202–209 (2019)
17. Gupta, B., Rahimi, S., Allam, V., Jupally, V.: Domino-effect free crash recovery for concurrent failures in cluster federation. In: Wu, S., Yang, L.T., Xu, T.L. (eds.) GPC 2008. LNCS, vol. 5036, pp. 4–17. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68083-3_4
18. ChkSim: A Distributed Checkpointing Simulator. https://dcomp.sor.ufscar.br/gdvieira/chksim/. Accessed 20 Oct 2019