Staggered Checkpointing and Recovery in Cluster Based Mobile Ad Hoc Networks

Parmeet Kaur Jaggi[1] and Awadhesh Kumar Singh[2]

[1] Department of Computer Science, Jaypee Institute of Information Technology, NOIDA, UP, India
parmeet.kaur@jiit.ac.in
[2] Department of Computer Engineering, National Institute of Technology, Kurukshetra, Haryana, India
aksinreck@rediffmail.com

Abstract. Checkpointing uses stable storage available in the distributed system for saving the consistent states of processes to which they can rollback at the time of recovery. But the checkpointing techniques for wired and cellular mobile systems are not trivially applicable to ad hoc networks as these networks have limited stable storage and wireless links are of low bandwidth. Moreover if synchronous checkpointing is employed, the processes contend for these limited resources at the time of checkpointing. This paper addresses the application of checkpointing to ad hoc networks and proposes a staggered approach to avoid simultaneous contention for resources. The staggering causes events, which would normally happen at the same time, to start or happen at different times. The proposed protocol does not need FIFO channels and logs minimum number of messages. It supports concurrent checkpoint initiation and successfully handles the overlapping failures in ad hoc networks.

Keywords: Checkpointing, Staggering, Concurrent initiators, Recovery, ad hoc networks.

1 Introduction

A mobile ad hoc network (MANET) is an autonomous collection of mobile nodes that communicate over relatively bandwidth constrained wireless links. The network topology is dynamic and decentralized; where all network activity including discovering the topology and delivering messages must be executed by the nodes themselves. Since the nodes communicate over wireless links, they have to contend with the effects of radio communication, such as noise, fading, and interference. In addition, the links typically have less bandwidth than in a wired network. The nodes have limited storage capabilities and typically no stable storage. The lifetime of a node may be determined by the battery life, thereby requiring the minimization of energy expenditure.

In such a scenario, the failure probability of the computing process increases greatly along with enlarging scale of the system. If a failure occurs in a computing process and there is not an appropriate method to protect it, more cost will be wasted

D. Nagamalai, E. Renault, and M. Dhanushkodi (Eds.): PDCTA 2011, CCIS 203, pp. 122–134, 2011.

for restarting the program. This need for reliability leads to the requirement of some fault tolerance method specifically designed for such networks. A major class of distributed systems uses checkpointing along with rollback recovery for providing fault tolerance. The work presented in this paper aims to present checkpointing as a fault tolerance approach in mobile ad hoc networks. We consider clustered mobile ad hoc networks in which nodes are partitioned into a number of virtual and disjoint groups called clusters. Under the cluster structure, mobile nodes are assigned a different function, such as cluster head (CH) or cluster member. One node in each cluster is chosen as the cluster head based on some criteria and the other members of the cluster use the stable storage at the cluster head for saving their checkpoints.

Three types of checkpointing protocols have been proposed in the literature,: *synchronous checkpointing*, where each process checkpoints simultaneously with every other process [1], *quasi- synchronous checkpointing*, where the communication history is piggybacked on each message, and each process checkpoints independently based on that information [1], and *asynchronous checkpointing*, where each process checkpoints independently, but the end result may be an inconsistent global state [1]. However, none considers contention.

The stable storage contention may not be a problem for asynchronous checkpointing as the processes take their checkpoints independently. Hence, checkpoints are often spaced apart on time axis. However, the synchronous checkpointing does not have this advantage. Therefore, it is advantageous for the synchronous checkpointing to stagger the checkpoints in order to avoid stable storage contention. The staggered checkpoints improve performance because, as the number of processes taking their checkpoints simultaneously and the checkpoint size grow, there is more contention present during synchronous checkpointing and thus more room for improvement when checkpoints are staggered [2]. When processes in a mobile ad hoc network's cluster contend for storage over limited bandwidth, staggering will bring a huge benefit to system performance.

Therefore the approach described in this paper uses a staggered checkpointing scheme adapted to the cluster based ad hoc environment. We aim to demonstrate that checkpointing can be a useful fault tolerance approach in ad hoc networks and staggering the checkpoints will give a boost to the performance. The next section discusses the work done in the area of checkpointing in MANETs. Further we put forth our system model. Subsequently, we describe our algorithm and its working. Then we present the recovery protocol. Lastly we conclude the presentation.

2 Related Work

Research on fault tolerance for the distributed systems has received tremendous interests in recent years. But these schemes can not be applied directly in ad hoc wireless networks due to the reason that there is no support of any static centralized administration and there are no fixed stable hosts or Mobile Support Stations.

2.1 Checkpointing MANETs

The work in [15] presents a cluster based checkpointing and rollback recovery scheme for ad-hoc wireless networks based on processes checkpointing and the

cluster-based multi-channel management protocol (CMMP).The network of mobile hosts is partitioned into several clusters. The mobile nodes act as cluster heads, gateways or ordinary members. Quasi synchronous checkpointing algorithm is employed along with pessimistic logging. Mobile hosts take checkpoints periodically managed by the local cluster head and log their output/input and messages related to the gateway.

The migratory services [16] model supports continuous and stateful client-service interactions in highly volatile ad hoc networks. The scheme uses context aware checkpointing to extend the primary backup approach for fault tolerance.

A checkpoint protocol for ad hoc networks has been proposed in [17]. Here, a checkpoint request message is delivered by flooding. State information of a mobile computer is carried by this message and stored into neighbor mobile computers. In the model of [18] the MANET is geographically partitioned into several disjoint and equal sized cluster regions. Each cluster is assigned a unique cluster id and has only one manager which is the one that can directly communicate with the adjacent managers For the recovery algorithm each manager must keep an ($n_{total\ h}$ * $n_{cluster\ h}$) dependency matrix where $n_{total\ h}$ is the total number of mobile hosts in the system and $n_{cluster\ h}$ is the total number of mobile hosts in its cluster

None of the above approaches addresses the problem of simultaneous access to stable storage and wireless channels by the checkpointing nodes.

2.2 Staggered Checkpointing

Staggered checkpointing has been proposed in literature for wired distributed systems. Chandy-Lamport algorithm [3] can stagger checkpoints when marker messages are forwarded, by the coordinator, to its neighbors only, which further forward the marker to their neighbors. However, the staggering vanishes in a completely connected topology where the coordinator directly forwards marker simultaneously to all processes. Based on Chandy-Lamport algorithm, two protocols, [4] and [5] have considered contention. Both allow processes to stagger their checkpoints and use either message logging or some form of additional synchronous checkpoints to guarantee a consistent state. A topology dependent algorithm to stagger a limited number of checkpoints is proposed in [4]. The work in [5] proposed an approach that could stagger all checkpoints. All three schemes work with single initiator and require FIFO message delivery.

The authors of [6] pointed out that the approach in [5] suffers from a major limitation. All messages must be logged in order to ensure global consistency. Hence, when the number of checkpointing processes in the system increases, the size of message log also increases dramatically. A heavy message log causes traffic for the stable storage leading to overall performance degradation. A solution to the problem has been put forth in [6]. As, processes contend for the stable storage; the availability of stable storage has been increased by using concurrent disks through a distributed RAID system [7].Since increasing the size of stable storage is not feasible in MANETs, we present an approach that can reduce the size of message log while staggering checkpoints.

Our algorithm is designed to work in MANETs with limited storage and non-FIFO channels. There can be concurrent initiators of the checkpointing process which will

speed up the process. The issue of concurrent initiations has been handled in literature. Most of the checkpointing algorithms[9][10][11][12], have assumed that the channels to be FIFO and have not considered the issue of contention.

We have used an approach similar to [10] to handle multiple initiations but our proposed protocol handles contention and does not need FIFO guarantee. Recently, two staggered quasi-synchronous checkpointing algorithms, as [13] and [14], have been presented. However, our checkpointing protocol is a staggered synchronous one.

3 System Model

Clustering approaches have been found suitable for large scale and high-density ad hoc network applications. A special node such as the cluster head can coordinate the message transmission and checkpointing of nodes in its cluster. We therefore apply our checkpointing algorithm to a cluster based network as in Fig 1. The nodes of the mobile ad hoc network are divided into clusters and one node is chosen as the Cluster Head (CH) in each cluster. For instance a node N identifies itself as a cluster head when it recognizes that it meets some predefined qualifying criteria. This criterion could be that a node having the lowest node ID within its one-hop neighborhood will become a CH. A CH and all its neighbors thus form a cluster.

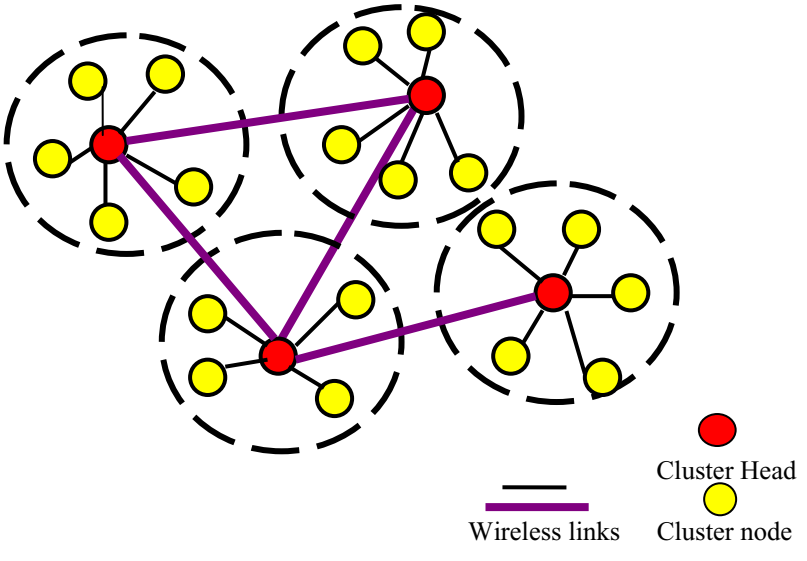

Fig. 1. System Model

The nodes which are chosen as the cluster heads, can then take part in the inter-cluster communication with the other CHs in their communication range. Ordinary nodes in each cluster can talk only within the cluster. The topology is such that any node can communicate with any other node in the cluster either directly or via the

CH. The CHs are assumed to have enough stable storage for saving the checkpoints of nodes in their cluster.

The processes in the system are fail-stop. The processes communicate with each other only by exchanging messages through an asynchronous and reliable channel with unpredictable but finite transmission delays. No message will be lost in the channel. The channel is non-FIFO and each message has a unique sequence number.

4 The Algorithm Concept

We present a staggering based synchronous checkpointing scheme adapted for handling the limited storage and bandwidth problems of MANETs. Controlled sender-based message logging is used, where only those messages are recorded in the message log that have been sent and yet not received at their respective destinations. We call them *in-channel* messages because these are the messages presently in the channel. If we could delay the recording of a local snapshot, the *in-channel* messages would get more propagation time and most of them would reach their respective destinations. Hence, the global snapshot collected by our approach represents a much more recent state of the system than what is collected without the proposed approach. In addition, the number of *in-channel* messages would be drastically reduced, which requires the small-sized message log to be maintained at sender. The messages are given sequence numbers to maintain consistency at the time of recovery.

Each CH has a distinct identifier (or id) and the hosts on joining a cluster also get an id. Each CH keeps a list, LOCAL of the active nodes in its cluster. The nodes in a cluster use the stable storage at the CH to save their most recent checkpoints.

Any CH may initiate the checkpointing and there can be multiple initiators of the checkpoint process. The initiator CHs are called leaders. The checkpoints are assumed to be sequenced so that all checkpoints with the same sequence number form a consistent state of the system. The further discussion of the algorithm describes the checkpointing process for a given checkpoint sequence number.

A leader CH after taking its own checkpoint sends a *take_chkpt* message, containing its own id, to other CHs in its transmission range and then initiates the checkpoint in its cluster. All the clusters which take a checkpoint in response to the message from the same leader form a group. Thus there will be as many groups formed in the system as there are concurrent leaders. Any CH on receiving the *take_chkpt* message for the first time saves the sender's id as its PARENT and the initiator's id as its LEADER. Every CH keeps a record of the recipients of its *take_chkpt* message by a 2D array, GLOBAL

A CH on receiving a *take_chkpt* message another time, sends a DENY message to the sender. It however keeps track of any concurrent leaders by a boundary-set data structure. If the initiator's id in any subsequent *take_chkpt* message is different from that saved in the LEADER variable at the CH, then the receiver CH saves this initiator's id in its boundary-set after sending a DENY message to the sender CH.

Thus the CHs form a forest of spanning trees in the system. Each leader is the root of a spanning tree and all CHs which take a checkpoint due to it belong to its spanning tree. A CH which sends a *take_chkpt* message to another for the first time is its parent in the spanning tree. The receiver of this *take_chkpt* message is its child

node. Within a cluster, a CH, after taking its own checkpoint, initiates the checkpoint by sending the *take_chkpt* message to its cluster nodes one by one in increasing order of their ids. This procedure continues till the last member of the cluster. Hence the nodes of the cluster take checkpoints at the CH in a staggered fashion.

When a leaf CH in the spanning tree has completed a checkpoint in its cluster, it sends an *ACK* message along with the boundary set to its parent in the spanning tree. This boundary-set is merged with the parent's boundary-set. After an intermediate CH in a spanning tree has received such *ACK* messages from its entire set of child CHs and has completed the checkpoint in its cluster, it sends an *ACK* message along with the boundary-set to its parent in the spanning tree

When the leader receives the acknowledgement of all its children CHs, it also knows the identifiers of other initiators in the system using boundary-set information it receives from the child CHs. The initiator then sends the *chkpt_taken* message to other initiators. When it has received similar messages from all concurrent initiators, it propagates a *chkpt_taken* message in the group formed by its child nodes to complete the checkpointing process for a given sequence number.

Thus our checkpointing protocol initiates the cluster heads at each level of a spanning tree in parallel. However, the nodes in a cluster are initiated sequentially. This approach has a two fold benefit. Firstly it removes the contention for CH storage and the wireless bandwidth as the checkpointing within each group is ordered sequentially for the nodes in the cluster while the checkpointing of different clusters can take place in parallel. Secondly, by delaying the checkpointing of some nodes, due to the sequence imposed upon them, some *in-channel* messages can reach their destinations, thereby reducing the size of message log.

5 The Working of Algorithm

The following messages have been used in the algorithm:

***take_chkpt<initiator CH id >*:** a CH sends this message to other CHs to take checkpoint, a CH also passes information about the initiator.

***ACK<boundary-set>*:** a CH sends this message to its PARENT after taking a checkpoint in its cluster carrying along with it the information it has about other concurrent leaders

***DENY*:** a CH which has already taken a checkpoint, on receiving a subsequent *take_chkpt* message sends *DENY* to sender

***chkpt_taken*:** a leader sends this message to other leaders after completing the checkpointing in its group

Any CH may initiate the checkpointing process by sending the *take_chkpt* message to the CHs in its transmission range. This message carries with it the sender's id so that any process receiving the *take_chkpt* message for the first time is included in the group of this initiator or leader. Since there can be multiple concurrent initiations, a CH receiving the *take_chkpt* message more than once replies to any subsequent senders with a *DENY* message. A CH which is not the leader replies with an *ACK*

message to its PARENT after completing the checkpointing in its cluster. The *ACK* message is appended with the information, if any, of other concurrent initiators.

For accomplishing the above, the algorithm uses the following data structures:

$GLOBAL_i$< CH id, flag>: is a 2D vector where each row denotes the recipients of the *take_chkpt* message; flag is 0 till the time an *ACK/DENY* is received back from the corresponding CH.

$LOCAL_i$: the set of active nodes in a cluster C_i // for simplicity assume nodes in a cluster are numbered 0,1,2…so on

PARENT: the CH which has sent the first *take_chkpt* message to CH_i

$LEADER_i$: the initiator CH due to which CH_i takes a checkpoint

Boundary-set_i: list of known concurrent initiators other than the LEADER

time_out: Boolean flag which denotes whether the waiting time for *ACK/DENY* messages has expired or not

Cluster_time_out: Boolean flag which denotes whether the waiting time for checkpoint of a node in the cluster has expired or not. If a node has not taken a checkpoint in this interval, CH can remove it from active nodes list.

Some member nodes may voluntarily or involuntarily disconnect from the MANET but we assume that a CH will not disconnect from the MANET. Every CH therefore maintains the status of each recipient CH of its *take_chkpt* message by the GLOBAL array .The flag bit in each row of the array is set to 1 only after receiving the *ACK/DENY* message from the corresponding CH. The recipients of the *take_chkpt* message keep a record of the sender by the PARENT identifier and the initiator by the LEADER identifier. If a CH does not respond within the *time_out* interval, its parent will again initiate that cluster to take the checkpoint. Since a CH does not disconnect, ultimately the flag bit for this CH at its parent will be set to 1. Within a cluster, some nodes may disconnect. Therefore a CH removes an inactive node from its current nodes list, $LOCAL_i$ if it does not take a checkpoint within a *cluster_time_out* interval.

6 The Algorithm

6.1 Pseudo Code

a) Initialization

for all *b*=0 to *m*-1,
    $GLOBAL_b$=NULL
    $LEADER_b$= NULL
    $PARENT_b$ =NULL
    boundary-set_b = NULL

Let CH_m, CH_n,… be various concurrent initiators

b) **PROCEDURE leader_chkpt(i) {**
//Each leader CH_i executes leader_chkpt(i)
(i) CH_i sends *take_chkpt(i)* to all CH_x in transmission range of CH_i
(ii) CH_i adds a record < CH_x, 0> to $GLOBAL_i$

(iii) CH_i calls Cluster_chkpt(i)
(iii) if(chk_global(i) ==TRUE) then send *chkpt_taken* message to boundary initiators
(iv)Wait for *chkpt_taken* message from other leaders
(v) If CH_i has has received *chkpt_taken* message from all members of its boundary-set, propagate *chkpt_taken* message in own group
}

PROCEDURE Cluster_chkpt(j) {
//Within a cluster j
(i) CH_j takes its checkpoint;
(ii) m=1
(iii) While there exists more elements in $LOCAL_j$
{ s=$LOCAL_j$[m] // member node of the cluster//
(iv) CH_j sends *take_chkpt* to node s;
(v) If Node s takes checkpoint then
{m++
Goto(iii)
}
else
{ if (cluster_time_out=1) then
{remove node s from $LOCAL_j$
goto (iii)
}
}
}
}

Procedure CH_checkpoint(i,j,k) {
//Each CH_k, which is not a leader, on receiving the *take_chkpt*(i) message from CH_j executes CH_checkpoint()
(i)If $LEADER_k$ <> NULL then
{send *DENY* to CH_j;
if $LEADER_k$ <>i then add i to < $boundary\text{-}set_k$ >
}
else
{ Set $LEADER_k$ = i
Set $PARENT_k$ = j
CH_k sends *take_chkpt(i)* to all CH_y // *y<>j*
Add a record < CH_y, 0> to $GLOBAL_k$
}
(ii) CH_k calls Cluster_chkpt(k)
(iii)if chk_global(k) is TRUE then
Send *ACK*<$boundary_k$> message to $PARENT_k$
}

PROCEDURE chk_GLOBAL(p) {
// A CH calls chk_GLOBAL to check if all child nodes have replied
While each flag($GLOBAL_p$)<> 1
{
if (time_out=0)
then
{ wait for *ACK* and *DENY* messages from CHs in $GLOBAL_p$
If CH_p receives a *DENY* message from CH_y, then set $GLOBAL_p$< CH_y, 1>
If CH_p receives an *ACK*<boundary-set> message from CH_y , then set $GLOBAL_p$< CH_y,1>, merge <boundary-set>$_y$ with <boundary-set>$_p$
}
}
if each flag($GLOBAL_p$)== 1 , *// all CHs for which CH_i was the PARENT have replied back//*
Return TRUE }

6.2 Proof of Correctness

Theorem 1. The checkpointing algorithm converges.

Proof: We prove this by contradiction. Suppose the checkpointing algorithm does not converge. Hence, there exists at least one cluster, say C_i that never finalizes its checkpoint. Now, there could be the following possibility: cluster C_i takes checkpoint under group leader, say C_l, as it receives *take_chkpt* message from another CH, say CH_j. Thus, if C_i is the last i.e. a leaf cluster of its group then it will send an *ACK* message to its parent CH_j; otherwise, it will send *take_chkpt* message to the next cluster of its group after taking its own checkpoint. The next process proceeds similarly and the last member i.e. leaf of the group sends an *ACK* message to its parent. Every parent ensures that each group member under its group has taken a checkpoint by checking the GLOBAL vector present with it. If some bit in the GLOBAL vector is zero corresponding to any particular cluster and *time_out* timer has expired then that parent will again initiate that cluster to take the checkpoint and corresponding messages will be replayed. Moreover within a cluster, a CH removes an inactive node from its current nodes list if it does not take a checkpoint within a cluster_time_out interval. Eventually, the checkpoint for the group is finalized. Therefore, there does not remain any cluster C_i which is not able to finalize its checkpoint. It is a contradiction. Thus, the checkpointing algorithm converges.

Theorem 2. No orphan messages can be generated in the system.

Proof: The algorithm requires controlled sender based message logging. Every message is logged at the sender at the time of sending. It is logged from the time it is sent till the time it is known to have been received at its destination. Thus the recording of the receipt of a message is preceded by the logging of its sending. Any message that is received has its determinant i.e. sending event logged at the sender. So, orphan messages cannot be generated by the system.

7 The Recovery Procedure

7.1 Data Structures Used

Let a process P_i be running on a node i
The following data structures are used for the recovery of process P_i :

SentSet$_i$: Every process maintains the set of all those processes to whom it has only sent messages to since the last checkpoint

RcdSet$_i$: Every process maintains the set of all those processes from which it has received messages since the last checkpoint

Countrcd$_i$: It keeps the count of messages P_i has received from any other processes P_j since the last checkpoint

nCountrcd$_i$: It keeps the count of messages P_i receives from other processes P_j during the recovery

rollback_status: Boolean variable (value 1 indicates that process is running recovery thread and value 0 indicates that process has completed the recovery successfully)

7.2 On Cluster Node Failure and Subsequent Recovery

We are using controlled sender based message logging. Only the *in-channel* messages are logged at the sender. Fig. 2 shows a failure at process P_i after it has sent messages m & m' to process P_k and received the message m" from P_j since its last checkpoint i2. Once the process P_i needs to recover after the failure, it rollbacks to its latest checkpoint, here i2, and replays the logged messages.

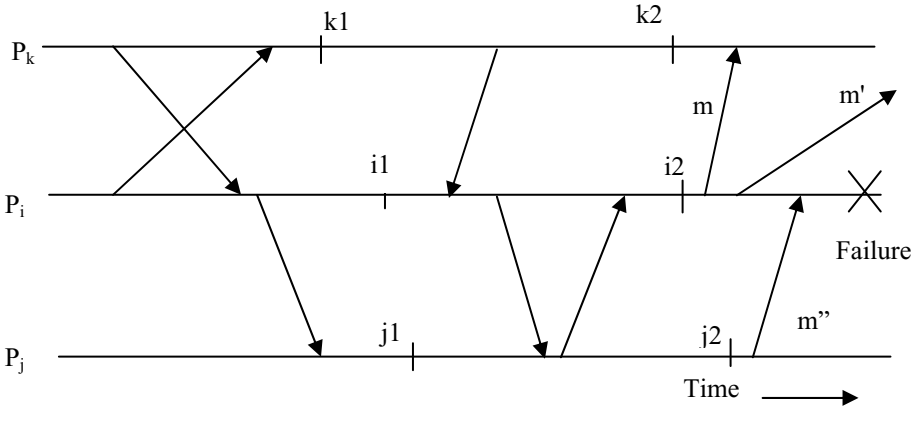

Fig. 2. Recovery of processes

Hence any process P_k that has received messages from P_i before P_i's failure need not rollback as it has either already received the message m from P_i before P_i's failure or will receive the in-channel message m' at the time of P_i's recovery. However, if a process P_j has also sent messages to P_i then P_j needs to rollback to the last checkpoint

, here j2, so that it can re-send any message m" earlier sent by P_j but un-received due to the rollback by P_i.

To avoid inconsistency in the system due to the recovery process, two approaches are possible. A non-blocking approach would allow processes like P_k to continue normal operation during recovery of any other process P_i to which no messages were sent during the last checkpoint interval. In such a case, all checkpoints of the same sequence number would form a consistent state of the system. A second blocking approach for recovery can require a recovering process P_i to send a RECOVERING message to all processes in $SentSet_i$ so that they do not advance their checkpoint till P_i has recovered. This will prevent orphan messages in the system. Upon the completion of recovery process, P_i can send a RECOVERED message to $SentSet_i$.

We assume the non-blocking approach described above for achieving consistent state in the system. No process is restarted from a state that has recorded the receipt of a message that no other process has recorded as received. If a process P_k has recorded the receipt of a message m between its checkpoint k2 and a later checkpoint k3, the sending of m shall be recorded by P_i between its checkpoint i2 and a later checkpoint i3.The recovery procedure never leads to an inconsistent state in the system.

Thus our algorithm provides an optimization by not requiring all processes in the system to roll back at the time of recovery. Only the processes in the $RcdSet_i$ are required to rollback to their latest checkpoint.

7.3 The Recovery Algorithm

Step1: upon restart after failure

(i) read $RcdSet_i$ from stable storage,
(ii) rollback to latest checkpoint
(iii) *Status* =Recovering;
(iv) *Rollback_status*=1;
(v) Send recover request to all processes those are in $RcdSet_i$

Step 2: if P_j receives a recovery request from P_i then

(i) P_j rollbacks to its latest checkpoint
(ii) P_j forwards the recovery request to all members of $RcdSet_j$;

Step 3: if multiple failure then

repeat step1 through step2 at each failed node P_k;
Step 4: P_i and each P_j replay the logged in-channel messages;
Step 5: if ($ncountrcdset_i$ = =$countrcdset_i$) then
$Status_i$= normal;
Rollback_status=0;
else initiate recovery again;

7.4 Proof of Consistent Recovery

Theorem 3. Recovery is consistent assuming reliable CHs and channels.

Proof: Upon recovery after failure, a process rollbacks to its latest checkpoint and resends any in-channel messages which are logged with it. Thus the messages to be sent by the process are eventually sent assuming reliable channels. Also the process sends a recovery request to processes in its *RecdSet* i.e. those processes from which it had received messages in the interval between its last checkpoint and failure. These processes rollback too and replay their messages. At any destination, messages can be placed in sequence and duplicates removed by using sequence numbers of the messages. Hence the recovery of nodes is consistent assuming reliable CHs and channels.

Theorem 4. The algorithm handles multiple failures.

Proof: Let, during the recovery of some failed process P_i at a node i some other process P_j has also failed at node j. When the failed process P_i initiates its recovery, it rolls back to its last checkpoint and then it sends the recovery request to all those processes from which it had received messages. On receiving the recovery request these processes will also roll back to their last checkpoints. Meanwhile, if some other process P_j fails, then P_j will also send the recovery request to the processes listed in its $RcdSet_j$. Thus, same procedure would be executed by P_j as for P_i .The two recovery threads would run concurrently. Thus, the protocol can successfully handle multiple failures.

8 Conclusions and the Scope of Future Work

Many researchers have concluded that, due to heavy message logging, the staggering should be discouraged in communication-intensive applications. Our technique subverts this inherent disadvantage of staggering. The small-sized message log makes our staggered approach a suitable candidate to checkpoint the communication-intensive applications too. Also, the more recent global snapshot collected by the proposed staggered protocol, jointly with the small-sized message log, eliminate any possibility of occurrence of the missing message problem. The algorithm will also scale up even when number of nodes is increased, since nodes are organized in a cluster based hierarchy. As the clusters can be initiated in parallel, the performance of the system does not degrade when new clusters are added to the system. The algorithm uses concurrent initiation and handles overlapping failures. Moreover, it does not need the channels to be FIFO. The rollback distance is limited to last checkpoint, even if frequency of application messages are less and processes are running in isolation. However, the algorithm has a scope for further improvement. We propose to utilize the inherent spatial and message redundancy present in the MANETs for better results in our future work.

References

1. Elnozahi, E.N., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys 34(3), 375–408 (2002)
2. Norman, A.N., Choi, S.E., Lin,C.: Compiler-generated staggered checkpointing. In: Proc. 7th ACM Workshop on Languages, Compilers, and Run-time Support for Scalable Systems LCR 2004, pp. 1–8 (2004)

3. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems 3(1), 63–75 (1985)
4. Plank, J.S.: Efficient checkpointing on MIMD architectures, Ph.D. dissertation, Dept. of Computer Science, Princeton Univ. (1993)
5. Vaidya, N.H.: Staggered consistent checkpointing. IEEE Transactions on Parallel and distributed Systems 10(7), 694–702 (1999)
6. Jin, H., Hwang, K.: Distributed checkpointing on clusters with dynamic striping and staggering. In: Jean-Marie, A. (ed.) ASIAN 2002. LNCS, vol. 2550, pp. 19–33. Springer, Heidelberg (2002)
7. Hwang, K., Jin, H., Ho, R., Ro, W.: Reliable cluster computing with a new checkpointing RAID-x architecture. In: Proc. 9th Workshop on Heterogeneous Computing HCW 2000, Cancun, Mexico, pp. 171–184 (2000)
8. Ahn, J.: An efficient algorithm for removing useless logged messages in SBML protocols. In: Chakraborty, G. (ed.) ICDCIT 2005. LNCS, vol. 3816, pp. 166–171. Springer, Heidelberg (2005)
9. Koo, R., Toueg, S.: Checkpointing and rollback-recovery for distributed systems. IEEE Transactions on Software Engineering SE-13(1), 23–31 (1987)
10. Spezialetti, M., Kearns, P.: Efficient distributed snapshots. In: Proc. 6th IEEE International Conference on Distributed Computing Systems, pp. 382–388 (1986)
11. Prakash, R., Singhal, M.: Maximal global snapshot with concurrent initiators. In: Proc. 6th IEEE Symposium on Parallel and Distributed Processing, pp. 344–351 (1994)
12. Mandal, P.S., Mukhopadhyay, K.: Concurrent checkpoint initiation and recovery algorithms on asynchronous ring networks. Journal of Parallel and Distributed Computing 64(5), 649–661 (2004)
13. Manivannan, D., Jiang, Q., Yang, J., Persson, K.E., Singhal, M.: An asynchronous recovery algorithm based on a staggered quasi-synchronous checkpointing algorithm. In: Pal, A., Kshemkalyani, A.D., Kumar, R., Gupta, A. (eds.) IWDC 2005. LNCS, vol. 3741, pp. 117–128. Springer, Heidelberg (2005)
14. Jiang, Q., Manivannan, D.: An optimistic checkpointing and selective message logging approach for consistent global checkpoint collection in distributed systems. In: Proc. IEEE International Parallel and Distributed Processing Symposium, pp. 1–10 (2007)
15. Men, C., Xu, Z., Li, X.: An Efficient Checkpointing and Rollback Recovery Scheme for Cluster-Based Multi-channel Ad Hoc Wireless Networks. In: Proc. of the 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2008), pp. 371–378. IEEE Computer Society, Washington, DC, USA (2008)
16. Riva, O., Nzouonta, J., Borcea, C.: Context-aware fault tolerance in migratory services. In: Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services (Mobiquitous 2008). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, article 22 (2008)
17. Ono, M., Higaki, H.: Consistent Checkpoint Protocol for Wireless Ad-hoc Networks. In: The 2007 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, pp. 1041–1046 (2007)
18. Juang, T.T., Liu, M.C.: An Efficient Asynchronous Recovery Algorithm In Wireless Mobile Ad Hoc Networks. J. of Internet Technology 4, 143–152 (2002)