Checkpointing Distributed Applications on Mobile Computers

R K Chauhan*, Parveen Kumar **, R K Jaryal***
*Deptt. of Computer Sc. & Applications, K.U., Kurukshetra (HRY), India-136119
**Deptt. of CSE, NIT, Hamirpur (HP), India-177005.
***Deptt. of Electrical Engg., NIT Hamirpur [HP]-177005.
pk223475@yahoo.com

Abstract

Checkpointing is an efficient way of implementing fault tolerance in distributed systems. Mobile computing raises many new issues, such as high mobility, lack of stable storage on mobile hosts (MHs), low bandwidth of wireless channels, limited battery life and disconnections that make the traditional checkpointing protocols unsuitable to checkpoint such systems. Checkpointing can be independent, synchronous, quasi-synchronous, or message logging based. In the present study, we review the concepts of Mobile Distributed Systems and Checkpointing. We discuss various aspects and types of checkpointing techniques. We specifically address the challenges and guidelines for designing checkpointing algorithms for mobile distributed systems.

Key Words: Fault Tolerance, Consistent Global State, Check-pointing, Mobile Systems.

1. Mobile Distributed Computing System

Recent years have witnessed rapid development of mobile communications. In the future, we will expect more and more people will use some portable units such as notebooks or personal data assistants. A mobile distributed computing system is a distributed system where some of the processes are running on mobile hosts (MHs). The term "mobile" implies able to move while retaining its network connections. A host that can move while retaining its network connections is an MH. An MH communicates with other nodes of the system via a special node called mobile support station (MSS) [1], [2]. An MH can directly communicate with an MSS (and vice versa) only if the MH is physically located within the cell serviced by the MSS. A cell is a geographical area around an MSS in which it can support an MH. An MH can change its geographical position freely from one cell to another or even to an area covered by no cell. At any given instant of time, an MH may logically belong to only one cell; its current cell defines the MH's location, and the MH is considered local to the MSS providing wireless coverage in the cell. An MSS has both wired and wireless links and acts as an interface between the static network and a part of the mobile network. Static network connects all MSSs. A static node that has no support to MH can be considered as an MSS with no MH. Critical applications are required to execute fault-tolerantly on such systems [1], [2]. The system model for supporting host mobility consists of two distinct set of entities: a large number of MHs and relatively fewer number of MSSs [Refer Figure 1.1]. All fixed hosts and the communication path between them constitute the static/fixed network. The fixed network connects islands of wireless cells, each comprising of an MSS and the local MHs. The static network provides

reliable, sequenced delivery of messages between any two MSSs, with arbitrary message latency. Similarly, the wireless network within a cell ensures FIFO delivery of messages between an MSS and a local MH, i.e., there exists a FIFO channel from an MH to its local MSS, and another FIFO channel from the MSS to the MH. If an MH did not leave the cell, then every message sent to it from the local MSS would be received in the sequence in which they are sent [1], [2]. Message communication from an MH MH_1 to another MH MH_2 occurs as follows. MH_1 first sends the message to its local MSS MSS_1 using the wireless link. MSS_1 forwards it to MSS_2, the local MSS of MH_2, via the fixed network. MSS_2 then transmits it to MH_2 over its wireless network. However, the location of MH_2 may not be known to MSS_1, therefore, MSS_1 may require to first determining the location of

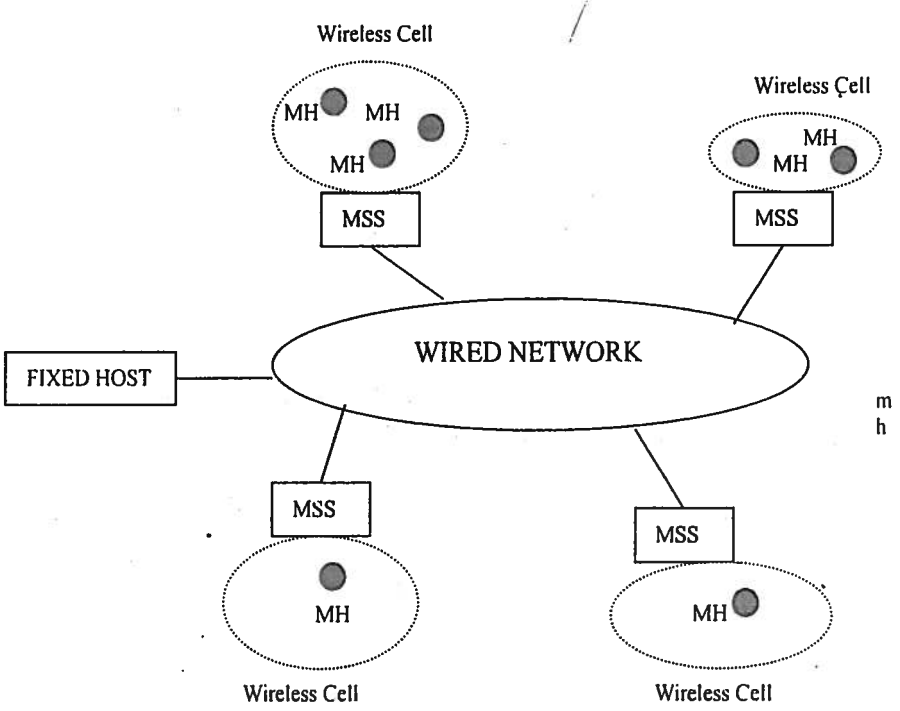

Figure 1.1 The system model for supporting host mobility

MH_2. This is essentially the problem faced by network
layer routing protocols [2].
Mobile Hosts often disconnect from the rest of the network. In our model, disconnection is distinct from failure. Disconnections are elective or volunteer by nature, so a mobile host informs the system prior to its occurrence and executes an application-specific disconnection protocol if necessary [2]. Disconnection can be voluntary on involuntary. We use the term "disconnection" to always imply a voluntary disconnection. We refer to an abrupt or involuntary disconnection as a failure.

2. Checkpointing

Fault tolerance can be achieved through some kind of redundancy. Redundancy can be temporal or spatial. In temporal redundancy, i.e., checkpoint-restart, an application is restarted from an earlier

checkpoint or recovery point after a fault. This may result in the loss of some processing and applications may not be able to meet strict timing targets. In spatial redundancy, many copies of the application execute on different processors concurrently and strict timing constraints can be met. But the cost of providing fault tolerance using spatial redundancy is quite high and may require extra hardware. Checkpoint-Restart or Backward error recovery is quite inexpensive and does not require extra hardware in general. Besides providing fault tolerance, checkpointing can be used for process migration, debugging distributed applications, job swapping, postmortem analysis and stable property detection [4]. There are two approaches for error recovery:

- Forward Error Recovery
- Backward Error Recovery

In **forward error recovery** techniques, the nature of errors and damage caused by faults must be completely and accurately assessed and so it becomes possible to remove those errors in the process state and enable the process to move forward [23]. In distributed system, accurate assessment of all the faults may not be possible. In backward error recovery techniques, the nature of faults need not be predicted and in case of error, the process state is restored to previous error-free state. It is independent of the nature of faults. Thus, backward error recovery is more general recovery mechanism [6]. There are three steps involved in backward-error recovery. These are:

- Checkpointing the error-free state periodically
- Restoration in case of failure
- Restart from the restored state

Backward error recovery is also known as checkpoint-restore-restart (CRR) or checkpoint-restart (CR). The checkpointing process is executed periodically to advance the recovery line. On failure, processes in distributed computation rolls back to latest checkpoint and then restart from the rolled back state.

A checkpoint is a local state of a process saved on stable storage. In a distributed system, since the processes in the system do not share memory, a global state of the system is defined as a set of local states, one from each process. The state of channels corresponding to a global state is the set of messages sent but not yet received. A global state is said to be "consistent" if it contains no orphan message; i.e., a message whose receive event is recorded, but its send event is lost [8], [12], [27], [28]. In Figure 1.2, the initial global state $\{C_{10}, C_{20}, C_{30}, C_{40}, C_{50}\}$ is consistent. It should be noted that initial global state is always consistent, because, it can not contain any orphan message. In Figure 1.2, the Global State $\{C_{11}, C_{21}, C_{31}, C_{41}, C_{51}\}$ is also consistent, because, it does not possess any orphan message. It needs to be noted that by definition, m_0 is not an orphan message. The Global State $\{C_{12}, C_{22}, C_{32}, C_{42}, C_{52}\}$ is inconsistent because it includes the orphan message m_8 [Refer Figure 1.2]. By definition, m_8 is an orphan message. To recover from a failure, the system restarts its execution from a previous consistent global state saved on the stable storage during fault-free execution. This saves all the computation done up to the last checkpointed state and only the computation done thereafter needs to be redone [8], [12], [27], [28].

After a failure, a system must be restored to a consistent system state. Essentially, a system state is consistent if it could have occurred during the preceding execution of the system

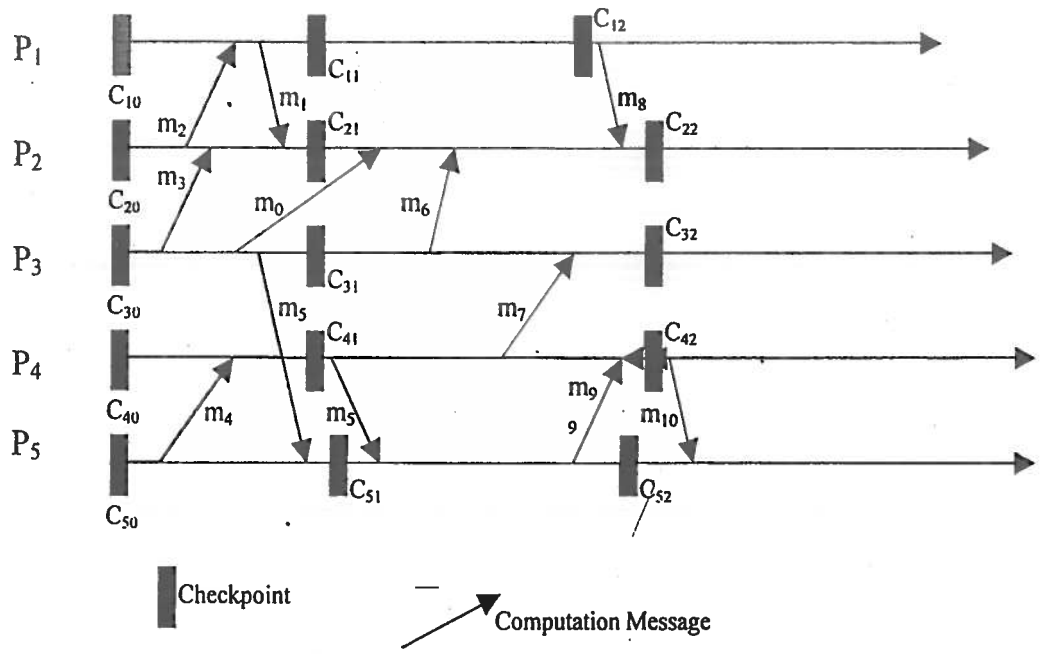

Figure 1.2 Consistent and Inconsistent Global States

from its initial state, regardless of the relative speeds of individual processes. This assumes that the total execution of the system is equivalent to some fault free execution [4], [8], [12]. It has been shown that two local checkpoints being causally unrelated is a necessary but not sufficient condition for them to belong to the same consistent global checkpoint. This problem was first addressed by Netzer and Xu who introduced the notion of a Z-path between local checkpoints to capture both their causal and hidden dependencies [15]. Considering a checkpoint and communication pattern, the rollback dependency trackability property stipulates that there is no hidden dependency between local checkpoints. To be able to recover a system state, all of its individual process states must be able to be restored. A consistent system state in which each process state can be restored is thus called a recoverable system state. Processes in a distributed system communicate by sending and receiving messages. A process can record its own state and messages it sends and receives; it can record nothing else. To determine a global system state, a process P_i must enlist the cooperation of other processes that must record their own local states and send the recorded local states to P_i. All processes cannot record their local states at precisely the same instant unless they have access to a common clock. We assume that processes do not share clocks or memory. The problem is to devise algorithms by which processes record their own states and the states of communication channels so that the set of process and channel states recorded form a global system state. The global state detection algorithm is to be superimposed on the underlying computation; it must run concurrently with, but not alter, this underlying computation [8].

The state detection algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds—a scene so vast that it cannot be captured by a single photograph. The photographers must take several

snapshots and piece the snapshots together to form a picture of the overall scene. All snapshots cannot be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed. Yet, the composite picture should be meaningful. The problem before us is to define meaningful and then to determine how the photographs should be taken [8].

The problem of taking a checkpoint in a message passing distributed system is quite complex because any arbitrary set of checkpoints cannot be used for recovery [8], [27], [28]. This is due to the fact that the set of checkpoints used for recovery must form a consistent global state. In backward error recovery, depending on the programmer's intervention in process of checkpointing; the classification can be:

- User-Triggered checkpointing
- Transparent Checkpointing

User triggered checkpointing schemes require user interaction and are useful in reducing the stable storage requirement. These are generally employed where the user has the knowledge of the computation being performed and can decide the location of the checkpoints. The main problem is the identification of the checkpoint location by a user [4].

The **transparent checkpointing** techniques do not require user interaction and can be classified into following categories:

- Uncoordinated Checkpointing
- Coordinated Checkpointing
- Quasi-Synchronous or Communication-induced Checkpointing
- Message Logging based Checkpointing

2.1 Uncoordinated Checkpointing

In uncoordinated or independent checkpointing, processes do not coordinate their checkpointing activity and each process records its local checkpoint independently [6]. It allows each process the maximum autonomy in deciding when to take checkpoint, i.e., each process may take a checkpoint when it is most convenient. It eliminates coordination overhead all together and forms a consistent global state on recovery after a fault [6]. After a failure, a consistent global checkpoint is established by tracking the dependencies. It may require cascaded rollbacks that may lead to the initial state due to domino-effect [12] [27], [28], [29]. It requires multiple checkpoints to be saved for each process and periodically invokes garbage collection algorithm to reclaim the checkpoints that are no longer needed. In this scheme, a process may take a useless checkpoint that will never be a part of global consistent state. Useless checkpoints incur overhead without advancing the recovery line [29].

2.2 Coordinated Checkpointing

In coordinated or synchronous checkpointing, processes take checkpoints in such a manner that the resulting global state is consistent. Mostly it follows two-phase commit structure [7], [8], [9], [12], [17], [25], [26]. In the first phase, processes take tentative checkpoints and in the second phase, these are made permanent. The main advantage is that only one permanent checkpoint and at most one tentative checkpoint is required to be stored. In case of a fault, processes rollback to last checkpointed state.

A straightforward approach to coordinated checkpointing is to block communications while the checkpointing protocol executes

[30]. A coordinator takes a checkpoint and broadcasts a request message to all processes, asking them to take a checkpoint. When a process receives the message, it stops its executions, flushes all the communication channels, takes a tentative checkpoint, and sends an acknowledgement message back to the coordinator. After the coordinator receives acknowledgements from all processes, it broadcasts a commit message that completes the two-phase checkpoint protocol. On receiving commit, a process converts its tentative checkpoint into permanent one and discards its old permanent checkpoint, if any. The process is then free to resume execution and exchange messages with other processes.

The coordinated checkpointing protocols can be classified into two types: blocking and non-blocking. In blocking algorithms, as mentioned above, some blocking of processes takes place during checkpointing [12], [26]. In non-blocking algorithms, no blocking of processes is required for checkpointing [7], [8], [9], [17]. The coordinated checkpointing algorithms can also be classified into following two categories: minimum-process and all process algorithms. In all-process coordinated checkpointing algorithms, every process is required to take its checkpoint in an initiation [8], [9]. In minimum-process algorithms, a subset of interacting processes is required to take their checkpoints in an initiation [7], [12], [17], [26].

To further reduce the system messages, needed to synchronize the checkpointing, loosely synchronized clocks are used [11], [29] . Neves et al. [16] gave a loosely synchronized coordinated checkpointing protocol that removes the overhead of synchronization. This approach assumes that the clocks at the processes are loosely synchronized. Loosely synchronized clocks can trigger the local checkpoints at all the processes roughly at the same time without a coordinator. After taking a checkpoint, a process waits for a period, which is sum of maximum time to detect a failure of other process in the system and the maximum deviation between clocks. It is assumed that all checkpoints belonging to a particular coordination session have been taken without the need of exchanging any message. If a failure occurs, it is detected within the specified time and the protocol is aborted.

Parveen Kumar et al [21] are the first to design blocking non-minimum-process algorithms that significantly reduce the blocking time as well as the number of useless checkpoints, where only minimum number of processes commit their checkpoints. A process takes its forced/induced checkpoint during checkpointing procedure only if there is a good probability that it will actually get the checkpoint request in the current initiation.

2.3 Quasi-Synchronous or Communication-induced checkpointing

Communication-induced checkpointing avoids the domino-effect without requiring all checkpoints to be coordinated [5], [11], [29]. In these protocols, processes take two kinds of checkpoints, local and forced. Local checkpoints can be taken independently, while forced checkpoints are taken to guarantee the eventual progress of the recovery line and to minimize useless checkpoints. As opposed to coordinated checkpointing, these protocols do no exchange any special coordination messages to determine when forced checkpoints should be taken. But, they piggyback protocol specific information [generally checkpoint sequence numbers] on each application message; the receiver then uses this information to decide if it should take a forced checkpoint. This decision is based on the receiver determining if past communication and checkpoint patterns can lead to the creation of useless checkpoints; a forced checkpoint is taken to break these patterns [5], [11], [29].

2.4 Message Logging based checkpointing protocols

Message-logging protocols are popular for building systems that can tolerate process crash failures [3], [11], [19], [29]. Message logging and checkpointing can be used to provide fault tolerance in distributed systems in which all inter-process communication is through messages. Each message received by a process is saved in message log on stable storage. No coordination is required between the checkpointing of different processes or between message logging and checkpointing. The execution of each process is assumed to be deterministic between received messages, and all processes are assumed to execute on fail stop processes.

When a process crashes, a new process is created in its place. The new process is given the appropriate recorded local state, and then the logged messages are replayed in the order the process originally received them. All message-logging protocols require that once a crashed process recovers, its state needs to be consistent with the states of the other processes [3], [11], [19], [29]. This consistency requirement is usually expressed in terms of orphan processes, which are surviving processes whose states are inconsistent with the recovered states of crashed processes. Thus, message- logging protocols guarantee that upon recovery, no process is an orphan. This requirement can be enforced either by avoiding the creation of orphans during an execution, as pessimistic protocols do, or by taking appropriate actions during recovery to eliminate all orphans as optimistic protocols do. Pradhan et al [22] have proposed that message logging based checkpointing is quite suitable for mobile environments.

2.5 Frequency of Checkpointing

A checkpointing algorithm executes in parallel with the underlying computation. Therefore, the overheads introduced due to checkpointing should be minimized. Checkpointing should enable a user to recover quickly and not lose substantial computation in case of an error, which necessitates frequent checkpointing and consequently significant overhead. The number of checkpoints initiated should be such that the cost of information loss due to failure is small and the overhead due to checkpointing is not significant. These depend on the failure probability and the importance of computation. For example, in transaction processing system when every transaction is important and information loss is not permitted, a checkpoint may be taken after every transaction, increasing the checkpoint overhead significantly [11].

2.6 Contents of a Checkpoint

The state of a process has to be saved in stable storage so that the process can be restarted in case of an error. The state/context includes code, data, and stack segments along with the environment and the register contents. Environment has the information about the various files currently in use and the file pointers. In case of message passing systems, environment variables include those messages which are sent and not yet received. The information that is necessary to resume a computation after it is pre-empted is called the context of that computation [11].

2.7 Overheads of a Checkpointing Algorithm

During a failure free run, every global checkpoint incurs coordination overhead and context saving overhead in a multiprocessor system. In parallel/distributed systems, coordination among processes is needed to obtain a consistent global state. Special messages and piggybacked information with regular messages are used to obtain coordination among processes. Coordination overhead is due to special control messages and piggybacked information. The book-

keeping operations necessary to maintain coordination also contribute to coordination overhead. The time taken to save the global context of a computation is defined as the context saving overhead. If stable storage is not available with every node in a multiprocessor system, the context is transferred over the network. Network transmission delay is also included in the overhead [11].

2.8 Application of Checkpointing

Besides its use to recover from failures, checkpointing is also used in debugging distributed programs and migrating processes in multiprocessor system. In debugging distributed programs, state changes of a process during execution are monitored at various time instances. Checkpoints assist in such monitoring. To balance the load of processors in the distributed system, processes are moved from heavily loaded processors to lightly loaded ones. Checkpointing a process periodically provides the information necessary to move it from one processor to another. With checkpointing, an arbitrary temporal section of a program's runtime can be extracted for exhaustive analysis without the need to restart the program from beginning [4], [11].

3. Hybrid Checkpointing Protocols

Minimum-process coordinated checkpointing is a suitable approach to introduce fault tolerance in mobile systems transparently. In this approach, some processes may not checkpoint for several checkpoint initiations as they are not part of minimum processes to checkpoint. In case of a recovery after a fault, such processes may rollback to far earlier checkpointed state and thus may cause greater loss of computation In coordinated checkpointing, where all processes checkpoint, the recovery line is advanced for all processes but the checkpointing overhead may be exceedingly high, especially in mobile environments; because, it will consume the scarce resources of mobile nodes even if they are not part of minimum processes to checkpoint. To optimize both, i.e., the checkpointing overhead and the loss of computation on recovery, a hybrid checkpointing algorithm, where an all-process checkpointing is forced after the execution of minimum-process coordinated checkpointing algorithm for a fixed number of times, is proposed. Thus, the Mobile nodes with low activity or in doze mode operation may not be disturbed in case of minimum-process checkpointing and the recovery line is advanced for each process after an all-process checkpoint [18], [19], [20].

In coordinated checkpointing, if a single node fails to checkpoint in an initiation, the whole checkpointing effort goes waste. It becomes difficult for multiple MHs to checkpoint synchronously due to disconnections and unreliable wireless channels. MHs are prone to frequent failures, which will require frequent rollback of all processes. In the literature, there are hybrid checkpointing protocols, where fixed hosts checkpoint synchronously and MHs checkpoint independently [10], [13], [14]. These schemes give MHs autonomy in taking checkpoints. An MH can recover independently by using its recent checkpoint and message log without forcing other nodes to rollback.

In deterministic systems, if two processes start in the same state, and both receive the identical sequence of inputs, they will produce the identical sequence outputs and will finish in the same state. The state of a process is thus completely determined by its starting state and by sequence of messages it has received [24], [29]. Anti-message is exactly like an original message in format and content except in one field, its sign. Two messages that are identical except for opposite signs are called anti-messages of one another. All messages

sent explicitly by user programs have a positive (+) sign; and their anti-messages have a negative sign (-). Whenever a message and its anti-message occur in the same queue, they immediately annihilate one another. Thus the result of enqueueing a message may be to shorten the queue by one message rather than lengthen it by one. Singh et al [24] proposed a minimum-process coordinated checkpointing algorithm for deterministic mobile distributed systems, where no useless checkpoint is taken, no blocking of processes takes place, and anti-messages of very few messages are logged during checkpointing. In other words, they eliminate useless checkpoints and blocking of processes both at the cost of logging anti-messages of selective messages only during checkpointing.

4. Challenges and Guidelines for Designing Checkpointing Algorithms for Mobile Distributed Computing Systems

The existence of mobile nodes in a distributed system introduces new issues that need proper handling while designing a checkpointing algorithm for such systems. These issues are mobility, disconnections, finite power source, vulnerable to physical damage, lack of stable storage etc. [1], [2], [7], [25]. The location of an MH within the network, as represented by its current local MSS, changes with time. Checkpointing schemes that send control messages to MHs, will need to first locate the MH within the network, and thereby incur a search overhead [2]. Due to vulnerability of mobile computers to catastrophic failures, disk storage of an MH is not acceptably stable for storing message logs or local checkpoints. Checkpointing schemes must therefore, rely on an alternative stable repository for an MH's local checkpoint [2]. Disconnections of one or more MHs should not prevent recording the global state of an application executing on MHs. It should be noted that disconnection of an MH is a voluntary operation, and frequent disconnections of MHs is an expected feature of the mobile computing environments [2]. The battery at the MH has limited life. To save energy, the MH can power down individual components during periods of low activity [2]. This strategy is referred to as the doze mode operation. The MH in doze mode is awakened on receiving a message. Therefore, energy conservation and low bandwidth constraints require the checkpointing algorithms to minimize the number of synchronization messages and the number of checkpoints.

The new issues make traditional checkpointing techniques unsuitable to checkpoint mobile distributed systems [1], [2], [7], [25]. Prakash-Singhal [25] proposed that a good checkpointing protocol for mobile distributed systems should have low memory overheads on MHs, low overheads on wireless channels and should avoid awakening of an MH in doze mode operation. The disconnection of MHs should not lead to infinite wait state. The algorithm should be non-intrusive and should force minimum number of processes to take their local checkpoints.

Minimum-process coordinated checkpointing is an attractive approach to introduce fault tolerance in mobile distributed systems transparently [25]. It avoids domino-effect, minimizes stable storage requirements, and forces only minimum interacting processes to checkpoint. To recover from a failure, the system simply restarts its execution from a previous consistent global checkpoint saved on the stable storage. But, it has the following disadvantages [7]. Some blocking of processes takes place or some useless checkpoints are taken. In order to record a consistent global checkpoint, processes must synchronize their checkpointing activities. In other words, when a process initiates checkpointing procedure, it asks all relevant processes to take their checkpoints. Therefore, coordinated checkpointing suffers

from high overhead associated with the checkpointing process. Sometimes, checkpoint sequence numbers are piggybacked along with computation messages. If a single process fails to checkpoint, the whole checkpointing effort of the particular initiation goes waste.

Acharya, A. [2] cast distributed systems with mobile hosts into a two tier structure: 1) a network of fixed hosts with more resources in terms of storage, computing, and communication, and 2) mobile hosts, which may operate in a disconnected, or doze mode, connected by a low bandwidth wireless connection to this network. They propose a two tier principle for structuring distributed algorithms for this model: To the extent possible, computation and communication costs of an algorithm is borne by the static network. The core objective of the algorithm is achieved through a distributed execution amongst the fixed hosts while performing only those operations at the mobile hosts that are necessary for the desired overall functionality.

In wireless cellular network, mobile computing based on a two-tier coordinated checkpointing algorithm reduces the number of synchronization messages.

5. Conclusions

In this paper, we have presented concepts of mobile distributed systems and checkpointing. We discussed various aspects and types of checkpointing techniques. We specifically addressed the challenges and guidelines for designing checkpointing algorithms for mobile distributed systems.

REFERENCES

1. Acharya A. and Badrinath B. R., "Checkpointing Distributed Applications on Mobile Computers," *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, pp. 73-80, September 1994.
2. Acharya A., "Structuring Distributed Algorithms and Services for networks with Mobile Hosts", *Ph.D. Thesis, Rutgers University*, 1995.
3. Alvisi, Lorenzo and Marzullo, Keith," Message Logging: Pessimistic, Optimistic, Causal, and Optimal", *IEEE Transactions on Software Engineering*, Vol: 24, No. 2, February 1998, pp. 149-159.
4. Avi Ziv and Jehoshua Bruck, " Checkpointing in Parallel and Distributed Systems", *Book Chapter from Parallel and Distributed Computing Handbook edited by Albert Z. H. Zomaya*, pp. 274-302, Mc Graw Hill, 1996.
5. Baldoni R., Hélary J-M., Mostefaoui A. and Raynal M., "A Communication- Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability," *Proceedings of the International Symposium on Fault-Tolerant-Computing Systems*, pp. 68-77, June 1997.
6. Bhargava B. and Lian S. R., "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems-An Optimistic Approach," *Proceedings of 17th IEEE Symposium on Reliable Distributed Systems*, pp. 3-12, 1988.
7. Cao G. and Singhal M., "Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing systems," *IEEE Transaction On Parallel and Distributed Systems*, vol. 12, no. 2, pp. 157-172, February 2001.
8. Chandy K. M. and Lamport L., "Distributed Snapshots: Determining Global State of Distributed Systems," *ACM Transaction on Computing Systems*, vol. 3, No. 1, pp. 63-75, February 1985.

9. Elnozahy E.N., Johnson D.B. and Zwaenepoel W., "The Performance of Consistent Checkpointing," Proceedings of the 11th Symposium on Reliable Distributed Systems, pp. 39-47, October 1992.
10. Higaki H. and Takizawa M., "Checkpoint-recovery Protocol for Reliable Mobile Systems," *Trans. of Information processing Japan*, vol. 40, no.1, pp. 236-244, Jan. 1999.
11. Kalaiselvi, S., Rajaraman, V., "A Survey of Checkpointing Algorithms for Parallel and Distributed Systems", *Sadhna,* Vol. 25, Part 5, October 2000, pp. 489-510.
12. Koo R. and Toueg S., "Checkpointing and Roll-Back Recovery for Distributed Systems," *IEEE Trans. on Software Engineering*, vol. 13, no. 1, pp. 23-31, January 1987.
13. Lalit Kumar, Parveen Kumar, R K Chauhan, "Logging based Coordinated Checkpointing in Mobile Distributed Computing Systems", *IETE Journal of Research*, vol. 51, no. 6, pp. 485-490, 2005.
14. Lalit Kumar, Parveen Kumar, R K Chauhan, "Message Logging and Checkpointing in Mobile Computing", *Journal of Multi-disciplinary Engineering Technologies,* Vol.1, No.1, 2005, pp. 61-66.
15. Netzer, R.H. and Xu,J ,"Necessary and Sufficient Conditions for Consistent Global Snapshots", *IEEE Trans. Parallel and Distributed Systems* 6,2, pp 165-169, 1995.
16. Neves N. and Fuchs W. K., "Adaptive Recovery for Mobile Environments," *Communications of the ACM*, vol. 40, no. 1, pp. 68-74, January 1997.
17. Parveen Kumar, Lalit Kumar, R K Chauhan, V K Gupta "A Non-Intrusive Minimum Process Synchronous Checkpointing Protocol for Mobile Distributed Systems" *Proceedings of IEEE ICPWC'-2005*, January 2005.
18. Parveen Kumar, Lalit Kumar, R K Chauhan, "A low overhead Non-intrusive Hybrid ynchronous checkpointing protocol for mobile systems", *Journal of Multidisciplinary Engineering Technologies*, Vol.1, No. 1, pp 40-50, 2005.
19. Parveen Kumar, Lalit Kumar, R K Chauhan, "Synchronous Cheekpointing Protocols for Mobile Distributed Systems: A Comparative Study", *International Journal of information and computing science*, Volume 8, No.2, 2005, pp 14-21.
20. Parveen Kumar, Lalit Kumar, R K Chauhan, "A Hybrid Coordinated
21. Checkpointing Protocol for Mobile Computing Systems", *IETE journal of research,* Vol 52, No. 2&3, pp 247-254, 2006.
22. Parveen Kumar, Lalit Kumar, R K Chauhan, "A Synchronous Checkpointing Protocol for Mobile Distributed Systems: A Probabilistic Approach", Accepted for Publication in *International Journal of Information and Computer Security, 2006.*
23. Pradhan D.K., Krishana P.P. and Vaidya N.H., "Recoverable Mobile Environment: Design and Trade-off Analysis," *Proceedings 26th International Symposium on Fault-Tolerant Computing*, pp. 16-25, 1996.
24. Pradhan D.K. and Vaidya N., "Roll-forward Checkpointing Scheme: Concurrent Retry with Non-dedicated Spares," *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 166-174, July 1992.
25. Pushpendra Singh, Gilbert Cabillic, "A Checkpointing Algorithm for Mobile Computing Environment", *LNCS, No. 2775*, pp 65-74, 2003.
26. Prakash R. and Singhal M., "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems," *IEEE Transaction On Parallel and Distributed Systems,* vol. 7, no. 10, pp. 1035-1048, October1996.
27. R K Chauhan, Parveen Kumar, Lalit Kumar, "A coordinated checkpointing protocol for mobile computing systems", To appear in *International Journal of information and*

computing science, Vol 9, No. 1, 2006.

28. Randall, B, " System Structure for Software Fault Tolerance", *IEEE Trans. on Software Engineering*, 1,2, 220-232, 1975.
29. Russell, D.L., "State Restoration in Systems of Communicating Processes", *IEEE Trans. Software Engineering*, 6,2. 183-194, 1980.
30. Elnozahy E.N., Alvisi L., Wang Y.M. and Johnson D.B., "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375-408, 2002.
31. Tamir, Y., Sequin, C.H., "Error Recovery in multi-computers using global checkpoints", *In Proceedings of the International Conference on Parallel Processing*, pp. 32-41, 1984

R K Chauhan*, Parveen Kumar **, R K Jaryal***
*Deptt. of Computer Sc. & Applications, K.U., Kurukshetra (HRY), India-136119
**Deptt. of CSE, NIT, Hamirpur (HP), India-177005.
***Deptt. of Electrical Engg., NIT Hamirpur [HP]-177005.
pk223475@yahoo.com