International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 06 Issue: 07 | July - 2022 Impact Factor: 7.185 ISSN: 2582-3930

A Review of Approaches for Compassionate Checkpointing with Mobile Computing Systems

Naheeda zaib
NIMS University Rajasthan Jaipur
NIMS School of Data Science and Engineering

ABSTRACT

A distribution system is a group of autonomous entities working together to address a challenge that cannot be addressed by any one of them alone. A distributed system called a mobile computing device (MCD) has certain processes that are executed on mobile nodes, whose position within the network shifts over time. Distributed mobile systems create new problems such as mobility, poor wireless channel bandwidth, disconnections, low battery life, and a lack of a steady, trustworthy store on mobile nodes. The issue of fault-tolerant computation in mobile distributed databases is discussed in this study. Checkpointing and roll-it-back recovery are the foundations for the procedures outlined.

1. INTRODUCTION

Due to their affordability, scalability, and ability to satisfy the requirements of high-performance computing, distributed computing and cluster computing are widely employed—the likelihood of failure exponentially with the number of components increases. Understanding the types of faults that can arise in these systems is crucial for fault tolerance. Permanent and transitory faults are the two basic types. Transient faults are brought on by changes in the environment, whereas permanent faults are brought on by long-term damage with one or more components. Component repair or replacement can fix permanent problems. Transient defects are hard to find and fix since they last for a brief period of time. As a result, fault tolerance becomes important, especially for temporary breakdowns in distributed systems. A system can accomplish tasks using fault-tolerant approaches, which include fault detection, fault localization, fault containment, and fault recovery. Component repair or replacement can fix permanent problems. Transient defects are hard to find and fix since they last for a brief period of time. As a result, fault tolerance becomes important, especially for temporary breakdowns in distributed systems. A system can accomplish tasks using fault-tolerant approaches, which include fault detection, fault localization, fault containment, and fault recovery. These systems include a range of computational, communication, and storage technologies. A system can experience a variety of fault causes, such as hardware failure, interference from the environment, software bugs, security breaches, and human mistakes. Permanent and transitory defects are the two categories into which faults may be divided. Faults that permanently harm a particular component of the system are known as permanent faults. Restoration of the damaged component and system reconfiguration is required for recovery from permanent problems. Transient defects are momentary and do not cause long-term harm. Because system reconfiguration is not required, recovery from transitory failures is easier than from permanent issues. Transient defects might dissipate without having any noticeable effects on the system, making it harder to identify them [8].

Through some form of redundancy, fault tolerance may be obtained. Redundancy may be geographical or temporal. When a defect occurs, an application is resumed using a previous checkpoint or recovery point in temporal redundancy, also known as checkpoint-restart. Applications could be unable to fulfil rigorous time requirements, and some processing may be lost as a result. Strict temporal limitations can be satisfied when there is spatial redundancy because several copies of the program run simultaneously on various processors. However, the expense of adopting spatial redundancy to provide fault tolerance is relatively expensive, and it can call for more hardware. In scientific and industrial applications, the program's execution must be halted and restarted from the beginning in the event of a transitory malfunction. As a reason, the large applications can only be finished if the system has a long enough fault-free period of time. If there are errors, the program's average execution time may increase exponentially over time. The main purpose of checkpointing is to prevent losing any useful processing that was completed prior to a problem. A program's state is periodically saved in a dependable storage media as part of checkpointing. The prior consistent condition is restored if a problem is found. Checkpointing allows a program's execution to be restarted in the event of a fault. This considerably reduces the amount of meaningful processing that is lost due to the problem. The average programmed execution with checkpointing only increases linearly with programmed length [8].

Backward error recovery, also known as checkpoint-restart, is often affordable and doesn't need additional hardware. Checkpointing may be utilized for process migration, distributed application debugging, task shifting, post-mortem analysis, and stable property identification, in addition to fault tolerance [95].

There are two methods for recovering from errors:

The type of mistakes and damage produced by failures must be thoroughly and precisely analysed in forward error correction approaches so that it is feasible to eliminate those errors from the system. The process's current condition allows it to proceed [70]. It might not be feasible to accurately analyse every failure in a distributed system.

The type of failures need not be predicted when using backward error recovery approaches, and in the event of an error, the process's state is returned to the prior error-free state. It doesn't depend on the type of fault. Backward error recovery is a more versatile recovery strategy as a result [14], [56].

Backward-error recovery consists of three phases. These are Restart from the restored state, Restoration in case of failure, and Periodic checkpointing of the error-free state.

Checkpoint-restore-restart (C.R.R.) or checkpoint-restart are other names for backward error recovery (C.R.R.). To move the recovery line forward, the checkpointing procedure is conducted frequently.

2. CHECKPOINTING

A checkpoint is indeed a local process state that is kept in secure storage to enable subsequent processing restart. Saving the status data is done by checkpointing because of a distributed system's processes. Share Memories defines a system's global state as a collection of individual process-specific local states.

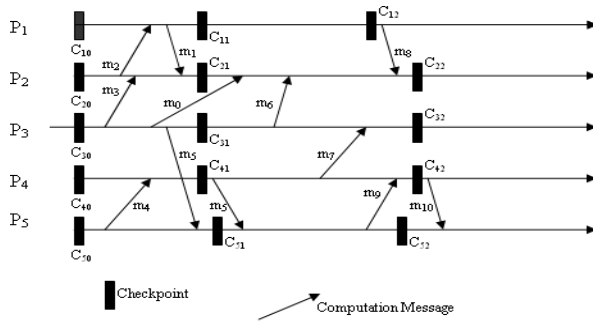

Figure 1.1 shows the global states' consistency and inconsistency

The collection of messages transmitted but that have not yet been received represents the state of pathways that corresponds to a global state. A message that was sent and recorded by the sender but could not be received and captured by the receiving process is referred to as being lost or in transit. An orphaned message is one whose send event was lost but whose receive event was recorded. If there are no orphan messages and all in-transit messages are logged, a global state is considered to be "consistent." The initial global condition of C10, C20, C30, C40, and C50 is consistent in Figure 1.1. Because it cannot include any orphan messages, the initial global status is always constant. Additionally compatible with the global state is C11, C21, C31, C41, and C51 because it is message-free and has no orphans. It should be noticed that just by nature, m0 is in message rather than an orphan message. Because it contains the orphan message m8, the global state "C12, C22, C32, C42, C52" is incoherent. M8 is indeed an orphan message by definition. The system resumes its execution after a failure from a prior consistent global state that was stored on the persistent storage throughout fault-free execution. The calculation up to the most recent checkpointed state is saved, and only the calculation performed afterward has to be restarted [8], [77], [78].

A system has to be brought back to a stable condition after a failure. Irrespective of the velocity vector of unit operations, any system state is essentially consistent if that would have happened throughout the operation of the plan that came before it from its beginning state. This is based on the presumption that the system would operate flawlessly throughout [8]. It has been demonstrated that for two local checkpoints to adhere to the same cohesive and comprehensive checkpoint, they must be causally unrelated to each other. In order to catch both their causation and hidden connections, Netzer and Xu [62] presented the idea of a Z-path between local checkpoints as the first solution to this issue. The rollback is based on checkpoints and communication patterns. It is a requirement of property that there be no covert relationship among local checkpoints [11]. A system state must be recoverable together with each of its separate process states. Thus, a recoverable process variable is a coherent system state whereby each process state may be restored.

A distributed system's processes interact with one another by exchanging messages. A process can only record its very own state and the communications it delivers and receives. A procedure that determines the overall system status. Other processes must cooperate with Pi by recording their respective local states and sending those records to Pi. It is impossible for all processes to record the local states at the exact same time. Unless they are able to use a shared clock, processes are assumed not to share memory or clocks. The challenge is to provide algorithms that

enable processes to record their own states as well as the states of channels of communication, forming a global system state from the collection of recorded processes and channel states. The underlying calculation is to be overlaid by the dynamic memory detection algorithm, which must operate concurrently with it without changing it [22].

A state detection method assumes the role of a team of photographers viewing a vast, dynamic picture that is too large to be recorded by a single shot, such as a sky full of migratory birds. To create a view of the overall scene, the photographer must take many photos and combine them. Due to synchronization issues, all pictures cannot be generated at the exact same moment. Additionally, the process being captured shouldn't be disturbed by the photographers. However, the whole image ought to have significance. We must first decide what is significant before deciding how to shoot the images [22]. Because any random collection of checkpoints cannot be utilized for recovery, setting a checkpoint in a message-passing distributed system is a challenging challenge [22], [77], [78]. This is because the collection of checkpoints in use for recovery has to create a stable global state.

Depending on the programmer's involvement throughout the checkpointing procedure, the categorization for backward error recovery might be:

Checkpoints Triggered by the User Checkpointing Transparency

Human-triggered checkpointing strategies necessitate user input while helping to lower the amount of reliable storage needed [27]. These are often used in situations where the user is aware of the calculation being done and has control over where the checkpoints should be placed. The user's ability to locate the checkpoint is the key issue.

The following categories can be used to group transparent checkpointing solutions that don't involve user interaction:

1.1 Uncoordinated Checkpointing

Processes need not synchronize their checkpointing activities in disorganized or independent checkpointing, and each process independently records its local checkpoint [14], [86], [96]. It gives each process the greatest degree of autonomy in determining when to take a checkpoint, allowing each process to do so whenever it is most practical. On recovery following a defect, it completely removes coordination overhead and creates a global sustainability state [14]. By monitoring the dependencies, a reliable global checkpoint is created following a failure. Due to the domino effect, it could need cascaded rollbacks that might return the system to its starting state [44], [77], [78].

Each process must have numerous checkpoints saved, and the garbage collection mechanism is regularly used to recover those checkpoints which are no longer required. An unnecessary checkpoint that would never be a part of the consistent global state may be taken by a process under this system. Checkpoints that are unnecessary cause overhead without moving the recovery line forward [27].

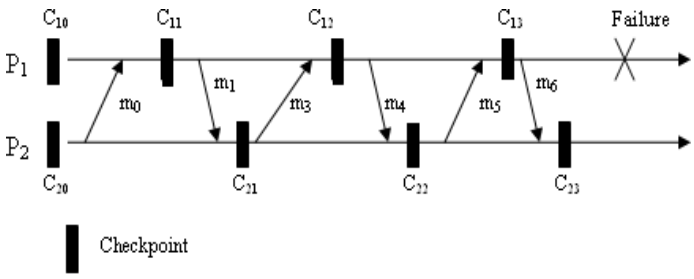

Fig. 1.2 The domino effect

The domino effect [Figure 1.2] is this strategy's biggest drawback. In this illustration, operations P1 and P2 had taken a series of checkpoints separately. There is only one consistent checkpoint for P1 and P2, the first one at "C10, C20," due to the interleaving of messages and checkpoints. P1 and P2 must thus restart the computation from the beginning once P1 fails [44]. It should be highlighted that orphan message m1 is the cause of the inconsistent global state "C11, C21." The orphan message m4 also causes the global state "C12, C22" to be inconsistent.

2.2 Co-ordinated checkpointing

When checkpoints are taken during coordination or synchronous checkpointing, the resultant global state is consistent. The commit structure is often two-phase [22], [28], [44]. Processes establish provisional checkpoints in the first stage, and in the

second stage, they are made permanent. The key benefit is that no more than one tentative checkpoint or one permanent checkpoint has to be stored. Processes roll back to the last checkpointed state in the event of a malfunction. A permanent checkpoint is irreversible. It ensures that the calculations required to arrive at the checkpoint state won't be repeated. However, a temporary checkpoint can be changed to a permanent one or reversed.

Blocking communication while the synchronized checkpointing protocol is running is a simple solution [88]. The coordinator performs a checkpoint and transmits a message to everyone's operations requesting that they do a checkpoint. Upon receiving the message, a process halts all executions, flushes all communication channels, executes a provisional checkpoint, and then replies to the coordinator with an acknowledgment message. The coordinator broadcasts a commitment message to end its two-phase checkpoint method after receiving recognitions from all processes.

When a process receives a commit, it turns its speculative checkpoint into a permanent one and, if any, discards its previous permanent checkpoint. After that, the process is free to continue running and communicate with other processes.

Blocking and non-blocking coordinated checkpointing techniques may be categorized into two groups. As was before established, checkpointing in blocking algorithms causes some process blocking [44], [88].

There is no need to block processes when using non-blocking algorithms [22], [28]. The two categories listed below can also be used to group coordinated checkpointing algorithms: minimal and total processes algorithms. Every process must take its checkpoint in an initiation when using all-process coordinated checkpointing techniques [22], [28]. Minimum interacting processes must take their checkpoints during an initiation in minimum-process algorithms [44].

2.3 Communication-Induced or Quasi-Synchronous Checkpointing

Without requiring that every checkpoint be coordinated, communication-induced checkpointing prevent the domino effect [12], [33], [55]. These protocols use local and forced checkpoints for their procedures. Locally checkpoints can be made on their own, but enforced checkpoints must be made to ensure the recovery line moves forward ultimately and to reduce pointless checks. In contrast to synchronized checkpointing, these procedures don't communicate specifically to coordinate when enforced checkpoints must be taken. However, they tack on protocol-specific data to every application message (often checkpoint sequence numbers), and the receiver utilizes this data to determine whether to take a forced checkpoint or not. The receiver's assessment of whether previous interaction and checkpoint tendencies can result in the establishment of pointless checkpoints informs this choice; a forceful checkpoint would then be implemented to disrupt these tendencies [27], [55].

2.4 Protocols for Message Logging-Based Checkpointing

For example, [3], [4], [5], [6], [9], [29], [30], [40], [74], [87], [90], [91], [92], [93] are message-logging protocols that are frequently used to create systems that can withstand process crash failures. In distributed systems where message-based inter-process communication is the only form of communication, message log and checkpointing could be employed to offer fault tolerance. A process logs every message it receives on stable storage in the message log. There is no need for coordination between messages log and checkpointing or even between checkpointing of various processes. All processes are considered to run on fail-stop processes, with each process' execution being presumed to be predictable between received messages.

A fresh process is started in the event of a process crash. The relevant recorded local state is transferred to the new process, and the logged message is then played it back in the sequence in which they were initially received by the process. When a wrecked process restarts, it must have a state that is compatible with both the state of all the other processes, according to all message-logging protocols [27], [98]. This requirement for consistency is usually described in terms of orphan operations, which are survivor processes with states that differ from the restored state of crashing processes. Therefore, message-logging techniques

ensure that no activity is an orphan upon recovery. This criterion can be implemented in one of two ways: either by taking suitable measures during recovery to destroy all orphans, as optimistic protocols would, or by preventing the production of orphans when in execution, as do pessimistic protocols. For mobile hosts, mobile support stations, and the home agent in an Ip Based environment, Bin Yao et al. [98] provide a receiver-based message recording protocol that ensures independent recovery. The use of checkpointing helps to reduce recovery latency and log size.

CHECKPOINTING PERSPECTIVES

3.1 Checkpoint Recurrence

The fundamental calculation is run concurrently with a checkpointing technique. Therefore, checkpointing overheads should be kept to a minimum. Checkpointing should make it possible for a user to recover fast and avoid losing a considerable amount of computation in the event of a mistake, which calls for frequent checkpointing and subsequently significant overhead. The number of checkpoints launched will be such that the overhead associated with checkpointing is negligible, and the cost of data losses caused by the failure is low. These are influenced by the likelihood of failure and the value of computation. A checkpoint could be performed after every operation in a transaction processing system, for instance, if every transaction is crucial and information loss is not allowed [42]. This dramatically increases checkpoint overhead.

3.2 Checkpoint Contents

In order to resume a process in the event of a mistake, its state must be preserved in a reliable storage location. Including the ambient and the contents of the registers, the state/context also contains portions of code, data, and the stack. The environment contains the file pointers and details of the different files that are currently in use. Environment variables include messages that have been delivered but have not yet been received in message-passing systems. The backdrop of that operation [42] is the knowledge required to continue an operation after it has been pre-empted.

3.3 Checkpointing Algorithm Overheads

Every global checkpoint during a failure-free run in a multiprocessor system results in coordination cost and context-saving overhead. To achieve a consistent global state in parallel/distributed systems, process coordination is necessary. To achieve process coordination, special messages and information that is piggybacked onto conventional communications are employed. Piggybacked information and specific control messages cause coordination overhead. The bookkeeping tasks required to keep coordination in place also add to its overhead. The overhead associated with context saving is the amount of time needed to save a computation's overall context. The context is transported via the network in a compute node if reliable storage is not present on every node. The overhead also includes the delay in network transmission [42].

3.4 Checkpointing in Practice

Checkpointing is used to migrate processes in multiprocessor systems and debug distributed programs in addition to recovering from errors. When debugging distributed applications, it's important to keep track of how a process's state changes over time. Checkpoints help with this kind of monitoring. Processes are transferred from processors that are significantly loaded to processors that are less loaded in order to balance the load on the distributed system's processors. A process can be moved from one computer to another by regularly checkpointing it [42]. Without having to start the program again from scratch, checkpointing allows for the extraction of any temporal segment of the runtime for thorough study [26].

3.5 Complementary Ideas

It becomes challenging to have a complete ordering of events when processes communicate with one another by exchanging messages because dependencies are established among the events of various processes. In order to obtain the expected occurrences in a distributed network, Lamport [52] suggested a relation termed "happened before" (denoted by). This relationship is transitive, antisymmetric, and irreflexive.

If events a and b are part of the same system and a happens-before b, then ab. If the event and is the sending of a message and event b is the receipt of that same message, then ab. If and only if a does not occur before b and b does not occur before a, two occurrences, a and b, are said to be contemporaneous. Local checkpoints are occasions where the condition of a process on a processor is

recorded at a certain moment. A number of the fellow checkpoint, one from each phase, make up a global checkpoint. If every event is part of a concurrent set, the global position is shown to be consistent. A set of local checkpoints—one from each process—that are all synchronous with one another constitutes a consistent global checkpoint. Resuming or recovering a calculation from a cohesive and comprehensive checkpoint is known as rollback recovery.

Calculation messages, or just messages, are the outputs of the underlying computation and are identified by the letters mi or m. Pi indicates the processes. The calculation between a process's ith and (i+1)th checkpoints, such as the ith checkpoint but excluding the (i+1) the checkpoint, is represented as the process' ith CI.

A course of action Pi is only directly dependent on Pj if m exists such that I Pi received m sent out by Pj (ii) Pi really hasn't reached a lasting checkpoint after receiving m (iii), and (iv) Pj has not reached a lasting checkpoint before sending m. A bit array of fixed length for n operations can hold direct dependencies at Pi. [Say ddvi[j]. Pi is implied to be directly reliant on Pj by the statement ddvi[j]=1 relationship between processes and minimal set computation [48], [64].

1. **CONNECTED WORK**

Several studies have been published on fault-tolerant checkpointing, according to a literature review. The bulk of them was developed by loosening up several of Chandy and Lamport's (1985) assumptions; the main objective of enhancing the previous extensions of Chandy & Lamport's (1985) algorithm was to reduce the operating costs of coordinating among activities in a multicore processor. To maintain consistent memory, a small number of techniques have been developed to checkpoint shared-memory multiprocessors. These algorithms essentially expand cache coherence protocols. These algorithms don't store context to disc and presume that the main memory is secure. Recently, methods for distributed shared memory systems have been put forth. For checkpoints in these systems, it is also crucial to maintain the cache cohesion of the virtual global memory. It is important to store main memory contents in a disc since physical memory is spread. Therefore, compared to shared-memory systems, contextual saving latency is larger. We also note that the majority of techniques make no assumptions on prior program structure knowledge intended for multiprocessor execution. Based on the presumption that hosts' locations in the network don't vary and their connectivity is constant in the absence of faults, techniques for distributed applications and their communications expenses have been designed. These presumptions are now invalid due to the development of smartphones. Furthermore, the power consumption of mobile hosts is strictly regulated, and the wireless connections that connect M.H.s to the local M.S.S.s have a certain amount of bandwidth.

One of the earliest non-blocking, all-process coordinated checkpointing algorithms for static nodes is the Chandy-Lamport [22] technique. This approach sends markers through every channel in the network, resulting in an O(N2) message complexity and necessitating FIFO channel ordering. Lai and Yang [50] suggested a method to loosen the FIFO assumption. When a process enters a checkpoint in this method, the piggybacks a signal onto the message that sends out through each channel. Before processing the message, the receiver looks just at the piggybacked flag to see whether a checkpoint is necessary. If so, a checkpoint is performed before the message is processed in order to prevent inconsistency. Each process must save the whole past messages on every route as part of local checkpoints in order to capture the channel information. All procedures must include checkpoints. An all-process non-blocking synchronous checkpointing technique with message complexity of O was proposed by Elnozahy et al. (N). They reduce the requirement for processes to be halted during checkpointing by identifying orphan messages using checkpoint sequence numbers. This strategy, however, necessitates communication between the initiator and every processing process. The processes that did not connect with one another during the last checkpointing period need not take fresh checkpoints in the method presented by Silva & Silva [85]. Both of these techniques [28], [85] presuppose that a notable initiator chooses the appropriate time to start the checkpointing operation. As a result, they experience the drawbacks of centralized algorithms, such as one-site failure, traffic jams, etc.

The method described by Leu & Bhargava [51] does not presume that the channels are FIFO, which is a prerequisite in [44] and is robust to many process failures. However, these two techniques [44] and [51] presume a sliding window type of scheme to address the message loss problem and do not take into account lost data in checkpointing and recovery. An algorithm was put out by Dang and Park [25] to deal with both lost and orphaned communications. A synchronized checkpointing strategy was initially suggested in the article [15]. It makes an overly restrictive presumption that almost all communications are atomic. The premise that all communications are atomic is relaxed by the minimal level coordinated checkpointing protocol introduced by Koo-Tong [44]. Both the number of checkpoints and synchronization messages is decreased. Only if it has communicated with Pi in the current CI the initiator process will send the checkpoint request. Similar to this, Pi will only make checkpoints demand to a process Pj if Pj has sent some m to Pi during the current CI. A synchronization tree is created in this manner, and the leaf node operations on a tree finally take their checkpoints. Due to movement, disconnections, and unstable wireless channels, coordinated checkpoint collection may take too long in mobile systems. Due to the processes' heavy stalling during checkpointing, the system's performance may suffer.

For mobile systems, Cao and Singhal [19] devised the minimum-process blocking technique. Comparing this approach to [44], blocking time is drastically decreased. For n processes, each process keeps track of its direct dependents inside a bit array of length n. The initiator process computes the smallest set by gathering all of the processes' direct dependence vectors. The checkpoint request and the minimum set are then communicated to all processes. A situation remained in the blocking phase during the time when it transmits its dependence vectors to the initiating processes and receives the minimal set. If a process falls under the minimum specified, it will reach its checkpoint. According to the algorithm [44], if some important process in initiation is unable to reach its checkpoint, the whole checkpointing procedure for that specific initiation is halted. An improved method to handle checkpointing failures was put forth by Kim and Park [45]. It enables some subtrees' new checkpoints to be committed. A procedure commits its preliminary checkpoint according to the method if none of the processes on which it transitively depends fail. For those operations that committed their checkpoints, the continuous recovery line is advanced. The initiator and any other processes that depend transitorily just on failing processes must abandon any tentative checkpoints. As a result, complete checkpointing abortion in the event of component failures is prevented. Loosely synchronized clocks are utilized [23], [63], [79], and [84] to further minimize the system messages required to synchronize the checkpointing. A coordinated checkpointing strategy that is loosely synchronized by Neves et al. [63] eliminates the overhead associated with synchronizing. According to this method, the processes' clocks are only weakly synced. Without a coordinator, clocks that are loosely synced may trigger all local checks at all of the processes nearly at the same time.

A procedure waits for a duration after setting a checkpoint, which is equal to the maximum amount of time needed to detect another program in the system failing and the maximum time allowed for clocks to differ. It is presumed that all checkpoints associated with a certain coordinating session have indeed been completed without the requirement of sending any messages. The protocol is terminated if a failure is discovered within the allotted period. A tool-aided method was developed by Sinha and Ren [75]. A technique for a timestamp-based checkpointing protocol's formal verification.

All of the aforementioned methods make an effort to minimize the overhead caused by coordinated checkpointing. The quantity of synchronization messages is kept to a minimum, checkpoint procedures are kept to a minimum [19], [44], and non-intrusive techniques are produced [22], [28]. The aforementioned algorithms are either non-intrusive or minimum-process.

The first minimal-process non-intrusive coordinated checkpointing mechanism for mobile distributed systems was proposed by Prakash and Singhal [72]. However, their algorithm could produce contradictions [19]. It was established in [19] that no minimal-process non-intrusive coordinated checkpointing technique exists. Therefore, some process blocking or pointless checkpoints are taken in minimal level

synchronized checkpointing algorithms [19], [44], [20], [48], and [64]. We may need to piggyback the numeric C.S.N. (checkpoints sequence number) on top of the regular messages in synchronized checkpointing protocols [20], [21], [28], [64], and [48]. For distributed systems, L. Kumar et al. [47] suggested some all non-intrusive checkpointing protocols in which just one bit is piggybacked onto regular communications. This is accomplished by adding extra overhead for vector transfer during checkpointing.

The idea of changeable checkpoints was introduced by Cao & Singhal [20] to achieve quasi in the minimal level approach. According to their methodology, an initiator, such as Pin, will only send a checkpoint demand to any process, such as Pj, if Pin has already received m from Pj within the current CI. If Pj has transmitted m to Pins in the current CI, Pj accepts its tentative checkpoint; if not, Pj determines that the request for a checkpoint is pointless. Similar to this, when Pj takes its provisional checkpoint, it broadcasts the request for a checkpoint to other processes.

The checkpointing tree is constructed as a result of continuing this procedure until the checkpoint request reaches all of the operations upon which the initiator transitively depends. When checkpointing, Pi could be required to take a checkpoint known as a mutable checkpoint if Pj sends m and Pj has already taken several checkpoints inside the current commencement before sending m. Pi's mutable checkpoint is worthless if it is not in the minimum threshold and is deleted on commit. In order to cut down on the number of pointless checkpoint requests, the enormous data structure M.R. [] is additionally connected with the checkpoint requests. Each procedure immediately sends the initiator its response.

Using the method suggested in [73], this algorithm [20] has already been constructed to support concurrent executions. The Cao-Singhal method [20] may result in inconsistencies during concurrent executions, as Ni et al. [61] have demonstrated. The algorithm suggested in [20] was revised by the authors [61] to support concurrent executions. In rare circumstances [48], the number of pointless checkpoints in [20] could be quite large.

By maintaining non-intrusiveness, L. Kumar et al. [48] & P. Kumar et al. [64] decreased the depth of the synchronization tree as well as the number of pointless checkpoints, though at the added expense of maintaining and gathering physical dependence vectors, computer technology the minimum set, and transmitting that on the deterministic system including the checkpoint proposal. In method [48], Pi analyzes m sent by Pj whether any of the following circumstances are true before transmitting the dependency vector and before getting the minimal set:

Since Pj is a direct dependant of Pi, Pj did not perform any checkpoints for such current commencement prior to transmitting m.

After transmitting m, Pj has made several long-term checkpoints.

Pi has already completed its generated checkpoint for the ongoing start.

For this initiation, Pi has already reached its induced checkpoint.

Since the most recent committed checkpoint, Pi did not send any messages.

Otherwise, before processing m, Pi performs its induced checkpoint, which is comparable to a mutable checkpoint.

Pi removes its preliminary checkpoint or changes any induced checkpoints it has into a tentative one if, after obtaining the minimal set, it discovers it was not a component of the minimum set. This approach does not create a checkpointing tree. If a process is in the minimal set when it receives the minimum set, the algorithm [64] instructs it to take its tentative checkpoint; else, it rejects the requests. If a process Pi is directly reliant on a process Pj and Pj is not included in the calculated minimum set; Pi transmits the checkpoint demand to Pj when Pi performs its tentative checkpoint. When Pi gets m via Pj, Pi only performs its triggered checkpoints before executing m if the following criteria are satisfied: I Pj checked a few things during the present commencement before sending m. (ii) Pi really hasn't taken any checkpoints during this initiation (iii) Pi has transmitted at most one message since the last permanent checkpoint. If Pi discovers that it isn't a member of the group upon commit, Pi dismisses its inspired checkpoints, if any, if it discovers it was not a part of the minimal set. In essence, the strategies suggested in [64] and [48] aim to reduce the amount of time a process may be compelled to wait before taking its induced/mutable checkpoint. The quantity of

pointless checkpoints is automatically decreased by shortening this duration.

The asynchronous checkpointing approach was put up by Acharya and Badrinath [1] for distributed systems on mobile distributed applications. For not using synchronized checkpointing for mobile systems, they provided the following justifications: Due to a Chandy Lamport [22] type of algorithm, M.H.s must respond to queries along every incoming connection, which results in 1) a high cost of identifying M.H.s and 2) non-availability of such local checkpoint of a detached M.H. throughout synchronized checkpointing. Every time a message receipt at a node is accompanied by a messaging broadcast, an M.H. is required by [1] to take its checkpoint. The number of local checkpoints would be equivalent to half the number of calculation messages if the transmitter and receiver messages are interleaved. This will probably result in extremely significant checkpointing overhead.

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] L. Alvisi, Hoppe, B., Marzullo, K., "Nonblocking and Orphan-Free message Logging Protocol," Proc. of 23rd Fault-Tolerant Computing Symp., pp. 145-154, June 1993.

[5] L. Alvisi, "Understanding the Message Logging Paradigm for Masking Process Crashes, "Ph.D. Thesis, Cornell Univ., Dept. of Computer Science, Jan. 1996. Available as Technical Report TR-96-1577.

[6] L. Alvisi and K. Marzullo, "Tradeoffs in implementing Optimal Message Logging Protocol", Proc. 15th Symp. Principles of Distributed Computing, pp. 58-67, A.C.M., June, 1996.

[7] Adnan Agbaria, William H Sanders, "Distributed Snapshots for Mobile Computing Systems", IEEE Intl. Conf. PERCOM "04, pp. 1-10, 2004.

[8] 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.

[9] A. Borg, J. Baumbach, and S. Glazer, "A Message System Supporting Fault Tolerance", Proc. Symp. Operating System Principles, pp. 90-99, ACM SIG OPS, Oct. 1983.

[10] Adnan Agbaria, William H. Sanders, "Distributed Snapshots for Mobile Computing Systems", Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications (Percom" 04), pp. 1-10, 2004.

[11] Baldoni R., Hélary J-M., Mostefaoui A. and Raynal M., "Rollback Dependency Trackability: A Minimal Characterization and its Protocol", Information and Computation, 165, pp. 144-173, 2003.

[12] 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.
[13] Bhagwat P., and Perkins, C.E., "A mobile Networking System based on Internet Protocol (I.P.)",USENIX Symposium on Mobile and Location-Independent Computing, August 1993.
[14] 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.
[15] G. Barigazzi and L. Strigni, "Application-Transparent Setting of Recovery Points", Digest of Papers Fault-Tolerant Computing Systems-13, pp. 48-55, 1983.
[16] Badrinath B. R, Acharya A., T. Imielinski "Structuring Distributed Algorithms for Mobile Hosts", Proc. 14th Int. Conf. Distributed Computing Systems, June 1994.

[17] Badrinath B. R, Acharya A., T. Imielinski "Designing Distributed Algorithms for Mobile Computing Networks", Computer Communications, Vol. 19, No. 4, 1996.
[18] Cao G. and Singhal M., "On coordinated checkpointing in Distributed Systems", IEEE Transactions on Parallel and Distributed Systems, vol. 9, no.12, pp. 1213-1225, Dec 1998.
[19] Cao G. and Singhal M., "On the Impossibility of Min- process Non-blocking Checkpointing and an Efficient Checkpointing Algorithm for Mobile Computing Systems," Proceedings of International Conference on Parallel Processing, pp. 37-44, August 1998.
[20] 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.
[21] Cao G. and Singhal M., "Checkpointing with Mutable Checkpoints", Theoretical Computer Science, 290(2003), pp. 1127-1148.
[22] Chandy K. M. and Lamport L., "Distributed Snapshots: Determining Global State of Distributed Systems," A.C.M. Transaction on Computing Systems, vol. 3, No. 1, pp. 63- 75, February 1985.
[23] F. Cristian and F. Jahanian, "A timestamp-based Checkpointing Protocol for Long-Lived Distributed Computations", Proc IEEE Symp. Reliable Distributed Systems, pp. 12-20, 1991.
[24] David R. Jefferson, "Virtual Time", A.C.M. Transactions on Programming Languages and Systems, Vol. 7, NO.3, pp 404-425, July 1985.
[25] Dang Y., Park, E.K. , "Checkpointing and Rollback- Recovery Algorithms in Distributed Systems", Journal of Systems and Software, pp. 59-71, April 1994.
[26] Dieter Kranzlmuller, Nam Thoai, Jens Volkert, "Error Detection in Large Scale Parallel Programs with Long runtimes, Future Generation Computer Systems 19, pp. 689- 700, 2003.
[27] Elnozahy E.N., Alvisi L., Wang Y.M. and Johnson D.B., "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," A.C.M. Computing Surveys, vol. 34, no. 3, pp. 375- 408, 2002.
[28] 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.
[29] Elnozahy and Zwaenepoel W, "Manetho: Transparent Roll-back Recovery with Low-overhead, Limited Rollback and Fast Output Commit," IEEE Trans. Computers, vol. 41, no. 5, pp. 526-531, May 1992.
[30] Elnozahy and Zwaenepoel W, " On the Use and Implementation of Message Logging," 24th int"l Symp. Fault-Tolerant Computing, pp. 298-307, IEEE Computer Society, June 1994.

[31] George H. Forman and John Zahorjan, "The Challenges of Mobile Computing", IEEE Computers vol. 27, no. 4, April 1994, pp. 38-47.
[32] Richard C. Gass and Bidyut Gupta, "An Efficient Checkpointing Scheme for Mobile Computing Systems", European Simulation Symposium, Oct 18-20, 2001, pp. 1-6.
[33] Hélary J. M., Mostefaoui A. and Raynal M., "Communication-Induced Determination of Consistent Snapshots," Proceedings of the 28th International Symposium on Fault-Tolerant Computing, pp. 208-217, June 1998.

[34] 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.
[35] Higaki H. and Takizawa M., "Recovery Protocol for Mobile Checkpointing", IEEE 9th International Conference on Database Expert Systems Applications, Viena, pp. 520-525, 1998
[36] Higaki H. and Takizawa M., "Checkpoint Recovery Protocol for Reliable Mobile Systems", 17th Symposium on Reliable Distributed Systems, pp. 93-99, Oct. 1998.
[37] Ioannidis, J., Duchamp, D., and Maguire, G.Q., "IP-based protocols for Mobile Internetworking", In Proc. of ACM SIGCOMM
Symposium on Communications, Architectures, and Protocols, pp. 235-245, September 1991.
[38] Johnson, D.B., Zwaenepoel, W., "Sender-based message logging", In Proceedings of 17th international Symposium on Fault-Tolerant Computing, pp 14-19, 1987.
[39] Johnson, D.B., Zwaenepoel, W., "Recovery in Distributed Systems using optimistic message logging and checkpointing. In 7th A.C.M. Symposium on Principles of Distributed Computing, pp 171-181, 1988.
[40] D. Johnson, "Distributed System Fault Tolerance Using Message Logging and Checkpointing," Ph.D. Thesis, Rice Univ., Dec. 1989.
[41] JinHo Ahn, Sung-Gi Min, Chong-Sun Hwang, "A Causal Message Logging Protocol for Mobile Nodes in Mobile Computing Environments", Future Generation Computer Systems 20, pp 663-686, 2004.
[42] Kalaiselvi, S., Rajaraman, V., "A Survey of Checkpointing Algorithms for Parallel and Distributed Systems", Sadhna, Vol. 25, Part 5, October 2000, pp. 489-510.
[43] Kistler, J., and Satyanarayana, M., "Disconnected Operation in the Coda file system", A.C.M. Trans. on Computer Systems 10, 1 (Feb. 1992).
[44] 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.
[45] J.L. Kim, T. Park, "An efficient Protocol for checkpointing Recovery in Distributed Systems," IEEE Trans. Parallel and Distributed Systems, pp. 955-960, Aug. 1993.
[46] Kyne-Sup BYUN, Sung_Hwa L.I.M., Jai-Hoon K.I.M., "Two- Tier Checkpointing Algorithm Using M.S.S. in Wireless

Networks", IEICE Trans. Communications, Vol E86-B, No. 7, pp. 2136-2142, July 2003.
[47] L. Kumar, M. Misra, R.C. Joshi, "Checkpointing in Distributed Computing Systems" Book Chapter "Concurrency in Dependable Computing", pp. 273-92, 2002.
[48] L. Kumar, M. Misra, R.C. Joshi, "Low overhead optimal checkpointing for mobile distributed systems" Proceedings. 19th IEEE International Conference on Data Engineering, pp 686 – 88, 2003.
[49] 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.
[50] T.H. Lai and T.H. Yang, "On Distributed Snapshots", Information Processing Letters, vol. 25, pp. 153-158, 1987.
[51] P.J. Leu and B.Bhargawa, "Concurrent Robust Checkpointing and Recovery in Distributed Systems", Proceeding Fourth Intl Conf. Data Engg. Pp. 154-163, Feb. 1988.
[52] L. Lamport, "Time, clocks and ordering of events in a distributed system" Comm. A.C.M., vol.21, no.7, pp. 558- 565, July 1978.
[53] Lalit Kumar, Parveen Kumar, R K Chauhan, "Pitfalls in Minimum-process Coordinated Checkpointing protocols for Mobile Distributed", ACCST Journal of Research, Volume III, No. 1, 2005 pp. 51-56.
[54] 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.
[55] Manivannan D. and Singhal M., "Quasi-Synchronous Checkpointing: Models, Characterization, and Classification," IEEE Trans. Parallel and Distributed Systems, vol. 10, no. 7, pp. 703-713, July 1999.
[56] Manivannan D., Netzer R. H. and Singhal M., "Finding Consistent Global Checkpoints in a Distributed Computation," IEEE Transactions on Parallel & Distributed Systems, vol. 8, no. 6, pp. 623-627, June 1997.

[57] Yoshifumi Manabe, "A Distributed Consistent Global Checkpoint Algorithm for Distributed Mobile Systems", 8th Int" l Conference on Parallel and Distributed Systems", pp. 125-132, 2001.
[58] Mannivannam, D., Singhal, M., "Failure Recovery based on Quasi-Synchronous Checkpointing in Mobile Computing Systems", In T.R. No. OSU-CISRC-7/96-TR-36, Dept of Computer and Information Science, The Ohio State University, 1996.
[59] Mannivannam, D., Singhal, M., "A Low overhead Recovery Techniques using Quasi Synchronous Checkpointing", Proc. 16th int "l conf. Distributed Computing Systems, pp 100-107, May 1996.
[60] Yoshinori Morita, Kengo Hiraga and Hiroaki Higaki, "Hybrid Checkpoint Protocol for Supporting Mobile-to-

Mobile Communication", Proc. Of the International Conference on Information Networking, 2001.
[61] Ni, W., S. Vrbsky and S. Ray, "Pitfalls in Distributed Nonblocking Checkpointing", Journal of Interconnection Networks, Vol. 1 No. 5, pp. 47-78, March 2004.
[62] 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.
[63] Neves N. and Fuchs W. K., "Adaptive Recovery for Mobile Environments," Communications of the A.C.M., vol. 40, no. 1, pp. 68-74, January 1997.
[64] 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.
[65] Parveen Kumar, Lalit Kumar, R K Chauhan, "A low overhead Non-intrusive Hybrid Synchronous checkpointing protocol for mobile systems", Journal of Multidisciplinary Engineering Technologies, Vol.1, No. 1, pp 40-50, 2005.
[66] Parveen Kumar, Lalit Kumar, R K Chauhan, "Synchronous Checkpointing Protocols for Mobile Distributed Systems: A Comparative Study", International Journal of information and computing science, Volume 8, No.2, 2005, pp 14-21.
[67] Parveen Kumar, Lalit Kumar, R K Chauhan, "A Hybrid Coordinated Checkpointing Protocol for Mobile Computing Systems", IETE Journal of research, Vol 52, No. 2&3, pp 247-254, 2006.
[68] 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.
[69] 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.
[70] 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.
[71] Pushpendra Singh, Gilbert Cabillic, "A Checkpointing Algorithm for Mobile Computing Environment", LNCS, No. 2775, pp 65-74, 2003.
[72] 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.
[73] Prakash R. and Singhal M., "Maximum Global Snapshot with Concurrent Initiations", Proc. Sixth IEEE Symp. Parallel and Distributed Processing, pp. 344-51, Oct. 1994.
[74] M.L. Powell and D.L. Presotto, "Publishing: A Reliable Broadcast Communication Mechanism", Proc. ninth Symp. Operating System Principles, pp. 100-109, ACM SIGOPS, Oct. 1983.

[75] Purnendu Sinha, Da Qi Ren, "Formal Verification of Dependable Distributed Protocols", Information and Software Technology, 45, pp. 873-888, 2003.
[76] Quaglia, F., Cipriani, R., Baldoni, R., "Checkpointing Protocols in Distributed Systems with Mobile Hosts: A Performance Analysis", IPPS/SPDP Workshop, pp. 742-755, 1998.

[77] Randall, B, "System Structure for Software Fault Tolerance", IEEE Trans. on Software Engineering, 1,2, 220- 232, 1975.
[78] Russell, D.L., "State Restoration in Systems of Communicating Processes", IEEE Trans. Software Engineering, 6,2. 183-194, 1980.
[79] Ramanathan, P. and K.G. Shin, "Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System", IEEE Trans. Software Engg., pp. 571- 583, June 1993.
[80] R K Chauhan, Parveen Kumar, Lalit Kumar, "A coordinated checkpointing protocol for mobile computing systems", International Journal of information and computing science, Accepted for Publication, Vol 9, No. 1, 2006.
[81] R K Chauhan, Parveen Kumar, Lalit Kumar, "Hybrid and intrusive synchronous checkpointing protocols for mobile distributed systems", Accepted for publication in ACCST Journal of Research, Volume IV, No. 4, 2006
[82] R K Chauhan, Parveen Kumar, Lalit Kumar, "Non-intrusive Coordinated Checkpointing Protocols for Mobile Computing Systems : A Critical Survey, ACCST Journal of Research, to be published in Volume IV, No. 3, 2006.
[83] R K Chauhan, Parveen Kumar, Lalit Kumar, "Checkpointing Distributed Applications on Mobile Computers", Journal of Multidisciplinary Engineering and Technologies, Vol. 2 No.1, Jan. 2006.
[84] Ssu K.F., Yao B., Fuchs W.K. and Neves N. F., "Adaptive Checkpointing with Storage Management for Mobile Environments," IEEE Transactions on Reliability, vol. 48, no. 4, pp. 315-324, December 1999.
[85] Silva, L.M. and J.G. Silva, "Global checkpointing for distributed programs", Proc. 11th symp. Reliable Distributed Systems, pp. 155-62, Oct. 1992.
[86] Storm R., and Termini, S., "Optimistic Recovery in Distributed Systems", A.C.M. Trans. Computer Systems, Aug, 1985, pp. 204-226.
[87] A.P. Sistla and J.L. Welch, "Efficient Distributed Recovery Using Message Logging", Proc. 18th Symp. Principles of Distributed Computing", pp 223-238, Aug. 1989.
[88] 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.
[89] Terakota, F., Yokote, Y., and Tokoro, M., "A Network Architecture providing host migration transparency", Proc, of ACM SIGCOMM 91, September 1991.

[90] S. Venkatesan and T.Y. Juang, "Efficient Algorithms for Optimistic Crash recovery", Distributed Computing, vol. 8, no. 2, pp. 105-114, June 1994.
[91] S. Venkatesan, "Message-Optimal Incremental Snapshots", Computer and Software Engineering, vol.1, no.3, pp. 211- 231, 1993.
[92] S. Venkatesan, "Optimistic Crash recovery Without Rolling back Non-Faulty Processors", Information Sciences, 1993.
[93] S. Venkatesan and T.T.Y. Juang, "Low Overhead optimistic crash Recovery", Proc. 11th Int. Conf. Distributed Computing Systems, pp. 454-461, 1991.
[94] Wada H., Yazawa, T., Ohnishi, T. and Tanaka, Y., "Mobile Computing Environment based on internet packet forwarding", Winter Usenix, Jan. 1993.

[95] Wang Y. M., Huang Y., Vo K.P., Chung P.Y. and Kintala C., "Checkpointing and its Applications," Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25),pp. 22-31, June 1995.
[96] Wood, W.G., "A Decentralized Recovery Control Protocol", 1981 IEEE Symposium on Fault-Tolerant Computing, 1981.
[97] Wang Y. and Fuchs, W.K., "Lazy Checkpoint Coordination for Bounding Rollback Propagation," Proc. 12th Symp. Reliable Distributed Systems, pp. 78-85, Oct. 1993.
[98] Bin Yao, Kuo-Feng Ssu & W. Kent Fuchs, "Message Logging in Mobile Computing", Proceedings of international conference on FTCS, pp 294-301, 1999.

[99] Yasuo Sato, Michiko Inoue, Toshimitsu Masuzawa, Hideo Fujiwara, "A Snapshot Algorithm for Distributed Mobile Systems" Proceedings of the 16th ICDCS, pp734-743,1996.