Global Journal of Computer Science and Technology
Volume 11 Issue 9 Version 1.0 May 2011
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
ISSN: 0975-4172 & Print ISSN: 0975-4350

The Performance of Soft Chekpointing Approach in Mobile Computing Systems

By Ruchi Tuli, Parveen Kumar
Singhania University

Abstract- Mobile computing raises many new issues such as lack of stable storage, low bandwidth of wireless channel, high mobility, and limited battery life. These new issues make traditional checkpointing algorithms unsuitable. Coordinated checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications since it avoids domino effects and minimizes the stable storage requirement. However, it suffers from high overhead associated with the checkpointing process in mobile computing systems. In literature mostly, two approaches have been used to reduce the overhead: First is to minimize the number of synchronization messages and the number of checkpoints; the other is to make the checkpointing process nonblocking. Since MHs are prone to failure, so they have to transfer a large amount of checkpoint data and control information to its local MSS which increases bandwidth overhead. In this paper, we introduce the concept of "Soft checkpoint" which is neither a tentative checkpoint nor a permanent checkpoint, to design efficient checkpointing algorithms for mobile computing systems. Soft checkpoints can be saved anywhere, e.g., the main memory or local disk of MHs. Before disconnecting from the MSS, these soft checkpoints are converted to hard checkpoints and are sent to MSSs stable storage. In this way, taking a soft checkpoint avoids the overhead of transferring large amounts of data to the stable storage at MSSs over the wireless network. We have also shown that our soft checkpointing scheme also adapts its behaviour to the characteristics of network.

Keywords: *Mobile distributed system, coordinated checkpointing, fault tolerance, Mobile Host.*

GJCST Classification: *C.1.3, C.2.1*

Strictly as per the compliance and regulations of:

The Performance of Soft Chekpointing Approach in Mobile Computing Systems

Ruchi Tuli[α], Parveen Kumar[Ω]

Abstract- Mobile computing raises many new issues such as lack of stable storage, low bandwidth of wireless channel, high mobility, and limited battery life. These new issues make traditional checkpointing algorithms unsuitable. Coordinated checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications since it avoids domino effects and minimizes the stable storage requirement. However, it suffers from high overhead associated with the checkpointing process in mobile computing systems. In literature mostly, two approaches have been used to reduce the overhead: First is to minimize the number of synchronization messages and the number of checkpoints; the other is to make the checkpointing process nonblocking. Since MHs are prone to failure, so they have to transfer a large amount of checkpoint data and control information to its local MSS which increases bandwidth overhead. In this paper, we introduce the concept of "Soft checkpoint" which is neither a tentative checkpoint nor a permanent checkpoint, to design efficient checkpointing algorithms for mobile computing systems. Soft checkpoints can be saved anywhere, e.g., the main memory or local disk of MHs. Before disconnecting from the MSS, these soft checkpoints are converted to hard checkpoints and are sent to MSSs stable storage. In this way, taking a soft checkpoint avoids the overhead of transferring large amounts of data to the stable storage at MSSs over the wireless network. We have also shown that our soft checkpointing scheme also adapts its behaviour to the characteristics of network.

Keywords- Mobile distributed system, coordinated checkpointing, fault tolerance, Mobile Host

I. Introduction

A mobile distributed system consists of both Mobile Hosts (MH) and static Mobile Service Stations (MSS). A set of dynamic and wireless communication links can be established between an MH and an MSS, and a set of high-speed communication link is assumed between the MSSs. An MSS may communicate with a number of MHs but an MH at a time communicates with only one MSS. An MH communicates with the rest of the system via the MSS it is connected to. Message transmission through wireless links takes an unpredictable but finite amount of time. Reliable message delivery is assumed during normal operation. The system does not have any shared memory or global clock [1]. Distributed computation in such mobile computing environment is performed by a set of processes executing concurrently on MHs and MSSs in the network. The processes communicate asynchronously with each other. A process experiences a sequence of state transitions during its execution and the atomic action which causes the state transition is called an *event*. The event having no interaction with another process is called an *internal event*, the message sending and receipt are *external events*. *Computation* is a sequence of state transitions within a process.

The diversity and flexibility introduced by mobile computing brings new challenges to the area of fault tolerance. Types of failures that were rare in the fixed environments are common with mobile hosts. Physical damage becomes much more probable, because mobile hosts are carried with the users while they move between sites. Mobile hosts can also be lost or stolen. Transient failures due to power or connectivity problems can be frequent events.

In this paper, we focus on checkpoint based recovery technique for mobile computing systems. A checkpoint protocol typically functions as follows : the protocol periodically saves the state of the application on stable storage. When a failure occurs, the application rolls back to the last saved state and then restarts its execution. Checkpoint protocols proposed in the literature are not suitable for mobile environments because of disconnections. Another problem is that these previously proposed protocols do not adapt their behaviour to the characteristics of the current network connection. If the network has a poor Quality of Service like small bandwidth and a high failure rate, the protocol should be able to trade off recovery time with operational costs.

II. Related Work and Problem Formulation

a) *Related Work*

The most commonly used technique to prevent complete loss of computation upon failure is Coordinated checkpointing [2], [3], [4], [5], [13]. In this approach, the state of each process in the system is periodically saved on the stable storage, which is called a checkpoint of the process. To recover from a failure,

About[α]- Research Scholar, Singhania University, Pacheri Bari (Rajasthan) India
About[β]- Professor, Meerut Institute of Engineering & Technology, Meerut (INDIA)

the system restarts its execution from a previous consistent global checkpoint saved on the stable storage. In order to record a consistent global checkpoint, processes must synchronize their checkpointing activities. In other words, when a process takes a checkpoint, it asks to all relevant processes by sending checkpoint requests to take checkpoints. Therefore, coordinated checkpointing suffers from high overhead. The protocol presented in this paper shows performance improvement over the work reported in [3], [4], [6], [7] & [8]. The protocol designed by Acharya and Badrinath [6] requires to create a new checkpoint whenever they receive a message after sending a message. Processes also have to create a checkpoint prior to disconnection. Pardhan et al. [7] proposed two uncoordinated protocols. The first protocol creates a checkpoint everytime when a process receives a message. The second protocol creates checkpoints periodically and logs all messages received. P. Kumar and R. Garg [11] proposed a hybrid scheme, wherein an all process checkpoint is enforced after executing minimum-process algorithm for a fixed number of times. In the first phase, the MHs in the minimum set are required to take soft checkpoint only. Soft Checkpoint proposed by them is stored on the disk of the MH and is similar to mutable checkpoint [8]. In the minimum process algorithm, a process takes its forced checkpoint only if it is having a good probability of getting the checkpoint request; otherwise, it buffers the received messages.

S. Kumar, R.K. Chauhan and P.Kumar [12] proposed a soft checkpoint approach in which a process in minset [ ] takes a soft checkpoint first and then soft checkpoint will be discarded, if it receives aborted message from the initiator. These soft checkpoints are saved on main memory of the mobile hosts [MHs], and then the soft checkpoint will be saved on the stable storage of MSS at a later time only if they receive the hard checkpoint request from the initiator. Their scheme requires low battery power of MHs, low checkpoint latency, low transmission cost, and low recovery time due to reduced disk accessed of MSS by the MHs. As soft checkpoint approach is less reliable, to make it reliable they transfer the soft checkpoint on stable storage

The protocols proposed in [3], [4] & [8] follow two-phase commit distributed structure. In the first phase processes take temporary checkpoints when they receive the checkpoint request. These tentative checkpoints are stored in stable storage of MSS. In the second phase, if an MSS learns that all the processes have taken the temporary checkpoints successfully, initiator MSS sends commit message to all the participating nodes. In these checkpoints an MH has to transfer a large amount of data to its local MSS over its wireless network which results in higher checkpoint latency and recovery time as transferring such temporary checkpoints on stable storage may waste a large amount of computation power, bandwidth, energy and time.

The protocol proposed by us creates a checkpoint whenever the local timer expires, and it only logs the unacknowledged messages at checkpoint time. Our protocol uses two types of checkpoints to recover from failure. The two previous protocols proposed in [6] and [7] always assume hard failures.

b) *Problem formulation*

In mobile distributed system multiple MHs are connected with their local MSS through wireless links. During checkpointing, an MH has to transfer a large of amount of data like control variables, register values, environment variable to its local MSS over the wireless network. So, it consumes resources like bandwidth, energy, time and computation power.

Mobile host failures can be separated into two different categories. The first one includes all failures that cannot be repaired; for example, the mobile host falls and breaks, or is lost or stolen. The second category contains the failures that do not permanently damage the mobile host; for example, the battery is discharged and the memory contents are lost, or the operating system crashes. The first type of failure will be referred to as hard failures, and the second type as soft failures. The protocol should provide different mechanisms to tolerate the two types of failures. The objective of the present work is to design a checkpointing approach which is suitable for mobile computing environment.

c) *Basic idea*

The basic idea of the proposed protocol is to use time to coordinate the creation of global states. Whenever, the local timer expires, the processes save their states periodically. Two distinct types of checkpoints are created by processes. The first checkpoint called the *soft checkpoint* saved locally in the mobile hosts to tolerate soft failures. The second type of checkpoints is *hard checkpoints* which is stored on stable storage of MSS and is used to recover from hard failures. Soft checkpoints are less reliable than hard checkpoints as the same can be lost with hard failures. But soft checkpoints cost much less than hard checkpoints. For different network configurations, the protocol saves distinct number of soft checkpoints per hard checkpoint. For a slow network, many soft checkpoints can be crated to avoid network transmissions

For a given network configuration, the protocol can exchange hard failure recovery time with performance costs. Hard failures are recovered with global states containing only hard checkpoints. The amount of rollback due to hard failures is small on average if the protocol creates hard checkpoints frequently, which causes the protocol to perform poor.

However, Soft checkpoints keep the system in running mode correctly while the mobile host is disconnected. In other words, a disconnected mobile host can be viewed as a host connected to a network with no bandwidth. In this case, the number of soft checkpoints per hard checkpoint is set to infinity, which means that all processes' states are stored locally. The local checkpoints are used to recover the mobile host from soft failures.

III. The Proposed Checkpointing Algorithm

a) *System Model*

The mobile environment model used in this protocol contains both fixed and mobile hosts interconnected by a backbone network. The fixed hosts are called MSS and mobile hosts are connected to MSS by wireless links. A MSS is connected to another MSS by wired network. The static network provides reliable and sequenced delivery of messages between any two MSSs. Similary wireless link between MSS and MH ensures FIFO delivery of messages. An MH can directly communicate with MSS only if the MH is physically located in that MSS. A cell is a geographical area around MSS which can have many MH. An MH can freely move from one cell to another and change its geographical position. At any instant of time, an MH can belong to only one cell. If an MH does not leave its cell, then every message sent to it from local MSS would be received in sequence in which it is sent.

b) *Algorithm Concept*

We assume that the protocol maintains a unique checkpoint number counter, *CkpNum*, at each process to guarantee that the independently saved checkpoints verify the consistency property. Whenever the process creates a new checkpoint, the value of *CkpNum* is incremented and is piggybacked with every message. The consistency property is ensured if no process receives a message with a $CkpNum_i$ larger than the current local *CkpNum*. If $CkpNum_i$ is larger than the local *CkpNum*, the process creates a new checkpoint before delievering the message to the application. The recoverability property is guaranteed by logging at the sender all messages that might become in-transit. These are the messages that have not been acknowledged by the receivers at checkpoint time. The sender process also logs the send and receive sequence number counters. During normal operation, these counters are used by the communication layer to detect lost messages and duplicate messages due to retransmissions. After a failure, each process resends the logged messages. Duplicate messages are detected as they are during the normal operation.

c) *Creation of a global state*

Whenever a mobile host moves out of the range of the cell or user turns off the network interface, it becomes disconnected. In a disconnected mode, the mobile host cannot access any information that is stored on a stable storage. Due to this reason, the protocol must be able to perform its duties correctly using local information. The protocol continues to save soft checkpoints to recover from soft failures. Two types of disconnections are considered. A *Temporary disconnection* allows the protocol to exchange few messages with stable storage just before the mobile host becomes isolated. Examples include the situations where communication layer informs the protocol when mobile host moves outside the range of cell or the boundary areas where signal strength becomes weaker. A *permanent disconnection* implies the case in which protocol is not able to exchange any messages with stable storage. Example includes when use unplugs the cable without turning off the application.

The creation of a new global state before disconnection is necessary for both the mobile host and the other hosts. This new global state is important because it prevents the rollback of work that was done while the mobile host was disconnected. If the new global state is not saved and another host fails after the disconnection, the application rolls back to the last global state that was stored (without warning the mobile host). The mobile host cooperates with the stable storage to create a new global state before disconnection. Just before the mobile host becomes isolated, the protocol sends to stable storage a request for checkpoint, and saves a new checkpoint of the process (hard or soft, depending on the network). Then the stable storage broadcasts the request to the other processes. Processes save their state as they receive the request. New global states can only be created before the mobile host detaches from the network if disconnections are orderly.

When the mobile host reconnects, the protocol sends a request to stable storage, asking for the current checkpoint number and the CN of the last hard global state. When the answer arrives, the protocol updates the local CN using the current checkpoint number. The protocol also creates a hard checkpoint if the mobile host has been isolated for a long time.

d) *Working of the Algorithm*

We illustrate the execution of the protocol with the help of following figure. This figure (Figure 1) represents the execution of three processes. Processes create their checkpoints at different instants, because timers are not synchronized. After saving its CkpNum checkpoint, process P1 sends message m1. When m1 arrives, process P3 is still in its CkpNum-1 checkpoint interval. To avoid a consistency problem, P3 first creates its CkpNum checkpoint, and then delivers m1. P3 also

resets the timer for the next checkpoint. Message m2 is an in-transit message that has not been acknowledged when process P2 saves its CkpNum checkpoint. This message is logged in the checkpoint of P2. Message m3 is a normal message that indirectly resynchronizes the timer of process P2. It is possible to observe in the figure the effectiveness of the resynchronization mechanism.

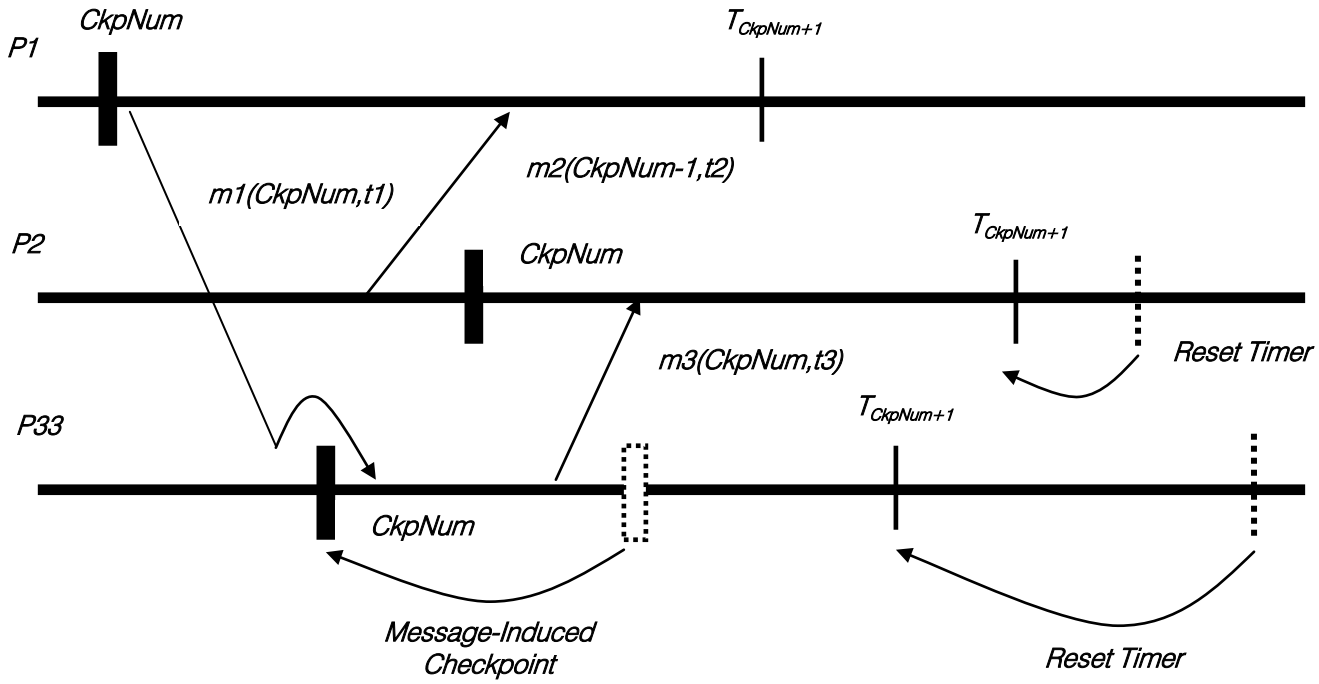

Figure 1:- Time-based soft checkpointing

e) *The Algorithm*

Following is the pseudocode of the algorithm. The algorithm uses the following local variables –

// S_i - Sender's Identifier

// $CkpNum_i$ – Current checkpoint number of the sender

// $timeToCkp_i$ – Time interval until next checkpoint

// msg_i – Message contents

i. *Message Receiving*

```
receiveMsg (Si, CkpNumi, timeTockpi, msgi)
if ((CkpNum=CkpNumi) and getTimeToCkp( ) > timeToCkpi))
                resetTimer (timeToCkpi);
else if (CkpNum<CkpNumi) {
        CreateCkp ( );
        resetTimer(timeToCkpi);
}
delieverMsgToApplication (msgi);
```

ii. *Application Process (At MH)*

```
createCkp ( ) :
        CkpNum : = CkpNum +1;
        resetTimer (T);
        if ((CkpNum mod maxsoft) = 0) sendCkpST
(getState ( ));
        else saveState (getState ( ), CkpNum);
```

C. Stable Storage (At MSS)

// The function arguments are same as in message receiving

```
receiveCkp (Si, CkpNumi, timeTockpi, statei)
        saveState (statei, CkpNumi);
        CkpNum : = max (CkpNum, CkpNumi);
        setFlag (CkpNumi, Si);
        if (row (CkpNumi) = 1) {
                CkpHard : = CkpNumi;
                garbageCollect (CkpHard);
        }
```

The functions given above are used to create a new checkpoint. Function *createCkp* is called to save a new process state. It starts by incrementing the *CkpNum*, and then it resets the timer with the checkpoint period. Next, the function determines if the checkpoint should be saved locally or sent to stable storage. The function *saveState* stores the process state locally, and the function *sendCkp* sends the process state to stable storage. The function *receiveCkp* is called by the stable storage to store newly arrived checkpoints. It first writes the received state to the disk, and then updates the local checkpoint counter. Then, it

determines if a new hard global state has been stored using a checkpoint table. The checkpoint table contains one row per *CkpNum*, and one column per process. The table entries are initialized to zero. An entry is set to one whenever the corresponding checkpoint is written to disk. The table only needs to keep one bit per entry, which means that it can be stored compactly. A new hard global state has been saved when all entries of a row are equal to one. The variable *CkpHard* keeps the checkpoint number of the new hard global state. The function garbageCollect removes all checkpoints with checkpoint numbers smaller than *CkpHard*.

IV. Adaptivity to Different Network Types

The protocol adapts its behavior to the characteristics of the network. If the network has a poor quality of service, the protocol saves many soft checkpoints before it sends a hard checkpoint to stable storage. The number of soft checkpoints stored per hard checkpoint is called *maxVal,* and it depends on the quality of service of the current network. The assignment of *maxVal* values to the different networks is made statically, and saved in a table. Table 1 gives two examples of possible assignments. The minimal quality of service corresponds to a disconnected mobile host. In this case, *maxVal* value is set to infinity, which means that only soft checkpoints are created. The minima *maxVal* column represents an assignment where hard checkpoints are created frequently, which guarantees a small re-execution time after a hard failure.

The maxima *maxVal* column corresponds to the opposite case. Application processes run on hosts that might be connected to different networks, each corresponding to a distinct *maxVal* value. This means that a global state can include both soft and hard checkpoints. To ensure that recovery is always possible, the protocol has to keep at each moment a global state containing only hard checkpoints. This global state is used to recover the application from hard failures. Otherwise, the domino effect [9] can occur, and recovery might not be possible. The protocol guarantees that new hard global states are saved by correctly initializing the *maxVal* table. The process that creates hard checkpoints less frequently is the one running in the host connected to the network with the worst quality of service. The protocol guarantees that a new hard global state is stored every time this process creates a hard checkpoint, by initializing the table in such a way that *maxVal* values are multiples of each other.

For example, if we have two processes P1 and P2 and the processes have *maxVal* value 2 and 4. This means that a new hard global state is created after every 2 and 4 soft checkpoints. Process P1 creates hard checkpoints whenever *CkpNum* is equal to 2, 4, 6, 8…. and process P2 creates whenever *CkpNum* is equal to 4, 8, 12, 16,….The protocol also keeps the last global state that was stored (which can include soft checkpoints) to recover from soft failures.

Table 1:- Creation of Hard Checkpoints

Quality Of Service	MaxVal	
	Minima	Maxima
Excellent	1	2
Good	2	8
Average	4	32
Poor	8	128
Disconnected	∞	∞

V. Comparison With the Related Work

In this section we compare our work with Acharya and Badrinath [6] and Pardhan et al. [7] since our work is very closely related to their work. The protocol designed by Acharya and Badrinath [6] requires to create a new checkpoint whenever they receive a message after sending a message. Processes also have to create a checkpoint prior to disconnection. Pardhan et al. [7] proposed two uncoordinated protocols. The first protocol creates a checkpoint every time when a process receives a message. The second protocol creates checkpoints periodically and logs all messages received. Also, the two protocols proposed in [6] and [7] always assume hard failures. These two algorithms have the following good features:

1. Only those processes that have received some message after sending a message, take checkpoints during checkpointing [6] or when process receives a message [7] thereby reducing the number of checkpoints to be taken.
2. Reductions in the number of checkpoints help in the efficient use of the limited resources of mobile computing environment.
3. Uses minimum interaction (only once) between the initiator process and the system of n processes and there is no synchronization delay.

However, the algorithms have a limitation too. Consider a system of n process distributed system. Let, the cost of sending a checkpoint request message from initiator to a single process be C_i. Hence, the checkpoint request cost, incurred by a single execution of the checkpointing algorithm, would be $(n-1)C_i$. Thus, the checkpoint request cost, incurred by k executions of the checkpointing algorithm, would amount to $k(n-1)C_i$. Therefore, the checkpoint request overhead, for applications involving large number of processes and running for longer durations, increases exponentially.

In the present work, we have attempted to eliminate above problem by using timer. It is a well-known fact that the use of timer eliminates extra

coordination messages [10]. A process takes checkpoint whenever its local timer expires. Moreover, only those processes take checkpoint, after expiry of their local timer, who have sent at least one message in the current checkpoint interval. Therefore, the number of processes taking checkpoint and, subsequently, the total number of checkpoints is significantly reduced. In addition, the use of timer removes need of the initiator process for sending the checkpointing request messages.

Our protocol creates a checkpoint whenever the local timer expires, and it only logs the unacknowledged messages at checkpoint time. Our protocol uses two types of checkpoints to recover from failure - *soft checkpoints* created and stored in MH to recover from soft failures and *hard checkpoints* created and stored at MSS to recover from permanent failures. Table 2 gives a comparison of our work on different parameters with the protocols proposed in [6] and [7].

Table 2 : Comparison with the related work

Parameters	Acharya and Badrinath [6]	Pardhan et al. [7]	Our Protocol
Creation of checkpoint	When a new message is received after sending a message	When process receives message	When local timer expires
No. of checkpoint phases	1	1	2
Failure assumed	Hard	Hard	Hard and Soft
Adaptable	No	Depend on wireless bandwidth	Vary with QoS of network
Coordination Method	Message based	Message based	Timer Based
Checkpoint Latency	High	High	Low
Transmission Cost	High	High	Low
Recovery Time	High	High	Less
CPU Overhead	High	High	High
Additional Hardware	Not Required	Not Required	Additional processor is required on MH
Main Memory requirement	Low	Low	High
Reliability	Low	Low	High
Efficiency	Low	Low	High
Suitability	For Large Systems	For Large Systems	For Large and small systems

VI. Conclusion

In our proposed approach, a have described a protocol that is able to save consistent recoverable global states. The process creates a new checkpoint whenever the local timer expires. The protocol stakes a soft checkpoint and saves it on the mobile host and later on before disconnection converts it to the hard checkpoint that is stored on MSS as soft checkpoint is less reliable. The protocol adapts its behavior to different types of networks by changing the number of soft checkpoints to be taken per hard checkpoint. When the mobile host is disconnected, the protocol creates soft checkpoints to recover from soft failures. The main features of our algorithm are: (1) it is non-blocking; (2) it is adaptive because it takes checkpointing decision on the basis of checkpoint sequence number; (3) it doesn't require tracking and computation of dependency information; (4) it doesn't require any control message because it uses timer to indirectly coordinate the creation of consistent global checkpoints and the local timers are not synchronized through control messages but by piggybacking control information on application messages.

References Références Referencias

1. C. Chowdhury, S. Neogy, "A Consistent Checkpointing- Recovery Protocol for Minimal number of Nodes in Mobile Computing System", *in International Conference on High Performance Computing*, pp-599-611, 2007.
2. Koo, R. and Toueg, S. (1987) 'Checkpointing and roll-back recovery for distributed systems', IEEE Trans. on Software Engineering, Vol. 13, No. I, January, pp.23–31.
3. G. Cao and M. Singhal, "On Coordinated Checkpointing in Distributed Systems," IEEE Trans. Parallel and Distributed System pp. 1213-1225, Dec. 1998.
4. E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel, [a]The Performance of Consistent Checkpointing,[o] Proc. 11th Symp. Reliable Distributed Systems, pp. 86-95, Oct. 1992.
5. R. Prakash and M. Singhal, "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems," IEEE Trans. Parallel and Distributed Systems, pp. 1035-1048, Oct. 1996.
6. Acharya, A. and Badrinath, B.R. Checkpointing distributed applications on mobile computers. In Proceedings of the Third International Conference on Parallel and Distributed Information Systems (Austin, Texas, Sep, 1994), pp 73-80.

7. Pradhan, D.K., Krishna, P., and Vaidya, N.H. Recovery in mobile environments: Design and trade-off analysis. In Proceedings of the 26th International Symposium on Fault-Tolerant Computing, (Sendai, Japan, June 1996), IEEE, pp. 16–25.
8. G. Cao and M. Singhal., "Mutable Checkpoints : A New checkpointing Approach for Mobile Computing Systems", In Proceedings of the IEEE Trans. Vol. 12, No. 2, pp-157-172, Feb. 2001
9. Randell, B. System structure for software fault tolerance. IEEE Trans. Softw. Eng. SE-1, 2 (June 1975), 220–232.
10. N. Neves, "Time-based coordinated checkpointing," Ph.D. dissertation, UIUCDCS-R-98-2054, University of Illinois at Urbana-Champaign, 1998.
11. P. Kumar and R. Garg, "Soft checkpointing based coordinated checkpointing protocol for Mobile Distributed Systems", International Journal of Computer Science Issues, Vol. 7, Issue 3, No. 5, May, 2010
12. S. Kumar, R.K. Chauhan and P. Kumar, "Reliable Soft-Checkpoint Based Fault Tolerance Approach for Mobile Distributed Systems", International Journal of Computer and Network Security", Vol. 2, No., June, 2010
13. Parveen Kumar, R K Chauhan, "A Coordinated Checkpointing Protocol for Mobile Computing Systems", International Journal of Information and Computing Science, Vol. 9, No. 1, pp. 18-27, 2006.

This page is intentionally left blank