A New Checkpointing Approach for Mobile Distributed System

Abstract

In this paper, we introduce a weighted checkpointing approach for the mobile distributed computing system (MDCS) that significantly reduces checkpointing overheads on mobile nodes. Checkpoint protocols proposed so far in the literature for MDCS are either coordinated, log based or quasi-synchronous. Coordinated checkpointing requires extra synchronization messages and may block the underlying computation. In quasi-synchronous approach processes have limited autonomy in checkpointing and all nodes need not to checkpoint concurrently as in the case of coordinated checkpointing. Also, only limited messages are logged as compared to log-based checkpointing where all messages are logged. Quasi-synchronous checkpointing protocols guarantee a consistent global state but the checkpointing frequency is governed by the message arrival pattern at various nodes and thus results in dynamic checkpointing overhead that cannot be anticipated. As a remedy for this, we propose a protocol that uses a hybrid of quasi-synchronous and message-logging approaches combined with the concept of weighted checkpoints. The weighted quasi-synchronous approach proposed by us requires no synchronization messages, reduces the checkpoints taken by the mobile nodes but requires logging of messages for mobile nodes. Simulation results and Comparison shows that the new approach is better than other proposed approaches.

1. Introduction

A Mobile Distributed Computing Systems generally has low bandwidth channels, mobile nodes with limited battery life, high fault rate and lack of stable storage on mobile nodes. These characteristics of MDCS make traditional checkpointing protocols inappropriate and necessitate the development of new approaches to checkpoint MDCS. The main challenges in taking a checkpoint of a MDCS are:

• **Vulnerability to physical damage:** Mobile hosts (MHs) are vulnerable to physical damage because they are exposed to outside universe and can be broken, stolen or the data stored can be damaged by radiations [1], [10]. Thus, in MDCSs the mobile host cannot be relied for any data that may be required after the failure.

• **Lack of stable storage at Mobile Hosts (MH):** In MH, the hard disk is either not present or if present, cannot be used to store MH's checkpoints due to its vulnerability to damage. So, the checkpoints of MHs are stored on its home or current Mobile Support Station (MSS) [1]. Frequent checkpointing of MHs puts heavy load of checkpoint transfer over wireless channels.

• **Finite power source:** Mobile hosts are powered by batteries that require periodical replenishment. To save the battery life a MH may have to frequently switch to low power usage mode i.e. Doze Mode of operation [6]. The checkpointing protocols that require frequent interaction with the nodes may not be suitable for MDCSs.

• **Low communication bandwidth:** The MHs are connected by low bandwidth wireless channels to MSSs [10]. A checkpointing protocol that requires large number of extra messages or requires large information to be piggybacked on messages may not be suitable for MDCSs.

• **Mobility and Disconnections:** Frequent disconnections and cell changes are feature of MDCSs [1], [10]. A checkpointing protocol that requires storing a new checkpoint on each move or disconnection of MH, may incur a high overhead.

• **High failure rate**: The mobile nodes are exposed to outside universe and have high failure rates as compared to their static counter parts, which are generally housed, in air-conditioned rooms. Thus, the recovery procedure should not impose higher overheads.

• **Resource Poorer**: The MHs are resource poorer than their static counter parts. Therefore, the checkpointing protocol should not add extra computation overheads on MHs. The MHs

may have limited memory and no stable storage. Thus, the techniques like memory checkpointing or local logging of non-deterministic events may not be feasible for MDCSs.

1.1 Motivation

Coordinated [5], [6], [17], [18], [25], quasi-synchronous [1] and log based [14], [15], 16] checkpointing protocols have been proposed for MDCSs. It has been advocated that coordinated checkpointing [9] is better for MDCSs [17] because the checkpointing activity can be controlled and no process is required to store more than two checkpoints at any time. But coordinated checkpointing protocols require additional synchronization messages [7], piggybacking of information on normal messages [9] and may also require blocking of underlying computation [13]. Moreover it requires that all nodes in a cell should take the checkpoints concurrently and transfer them to their current MSS. This will incur high concurrent load on the wireless network. If the density of MHs in a cell is more, it may give an illusion of blocking the MHs. The MHs may also be disconnected and may not be available for consistent checkpoint, which can put the protocol in infinite wait state. This is taken care by a checkpoint of the MH just before disconnection [6], [26]. Time based coordinated checkpointing [2] assumes loosely synchronized clocks at different nodes.

Message logging protocols can be pessimistic, optimistic, or causal [24], [27], [29]. Whereas, pessimistic or causal message logging protocols may slow the computation, the optimistic logging protocols may create orphans. In log-based approaches, the processes need to log the determinants for the messages synchronously or asynchronously. In case of MDCSs, this will incur high load on wireless channels, as MHs has no reliable stable storage.

In quasi-synchronous checkpointing there is no control over the number of checkpoints that are taken by a process. In the worst case, the number of checkpoints taken by a process may be equal to the number of messages in the distributed system [1]. These checkpoints are transferred to MSS resulting in high overhead on wireless channels.

Quasi-synchronous checkpointing may be attractive for MDCS if somehow the number of checkpoints taken by a mobile node can be reduced. This is not simple because any arbitrary reduction in checkpoints may lead to domino effect [22] at the time of recovery, which is highly undesirable.

Observation I: In quasi-synchronous checkpointing, the number of checkpoints taken by a mobile node increase very rapidly as the message arrival rate increases, whereas only one out of these is used for recovery in case of a fault.

Based upon this observation and the observations given in section 4, we propose a modified quasi-synchronous checkpointing scheme where the checkpointing activity at MHs is reduced. This reduction is at the cost of message logging for MHs only. It also gives some autonomy to MHs in taking a checkpoint, as the MHs may not be able to checkpoint concurrently [12]. We have correlated the possibility of a checkpoint to be the part of a recovery line [22] with a weight function that indicates whether the checkpoint should be taken or not. Thus, in our scheme, there are two kinds of checkpoints: actual and skipped or dummy. In case of a fault if a skipped checkpoint is part of the consistent global state and is required for recovery it can be constructed using message logs [12], [15], [21], [23] maintained by MSSs.

The probability that a checkpoint will be needed during recovery depends on many parameters such as the message arrival rate, fault rate, number of messages sent by a process before it takes a checkpoint and the communication pattern. Our protocol uses these parameters to calculate the weight of a checkpoint that decides whether to take/ skip the checkpoint. We study the effect of different parameters on the performance of the protocol. The simulation studies show that in many situations substantial gains in performance can be achieved using the proposed scheme.

1.2 Contribution

In this paper, we investigate the possibility of reducing the number of checkpoints taken by a quasi-synchronous protocol. We first study the effect of various parameters such as message sending rate, fault rate etc. on the number of checkpoints taken by the protocol. Based on our observations, we propose a weighted quasi-synchronous checkpointing protocol that is a hybrid of quasi-synchronous and log based approach. The resulting protocol requires no synchronization messages, reduces checkpointing overhead on MHs and gives them some autonomy at the cost of storing message logs for MHs. We have designed recovery protocol for the proposed checkpointing protocol. We have simulated the protocol and found that the performance of proposed protocol is better than the existing protocols. The hybrid protocol has been designed for a specific quasi-synchronous protocol [1] but the same approach can be extended to design a new hybrid checkpointing protocol using other quasi-synchronous checkpointing protocols with minor modifications.

2. System Model

A mobile distributed computing system consists of a number of MHs and relatively less MSSs [1]. All the MSSs are connected by wired static network. We assume that the static network provides reliable FIFO delivery of messages [5], [6]. A cell is a geographical area around MSS and the MSS can support a MH only within this area. While in the cell of a MSS, the MH communicates with it by a reliable FIFO wireless channel [5], [6]. The messages are delivered over wired or wireless channels with arbitrary but finite delay [6]. All processes are fail-safe and they do not share a common memory or clock [6]. The synchronization takes place only through message passing. All cells collectively constitute the mobile portion of the distributed system called *mobile network*.

The distributed computation consists of "N" processes denoted by P_0, P_1, P_2, …, P_{N-1} running concurrently either on MHs or on MSSs [6]. The computation is asynchronous i.e. each process has its own speed.

2.1 Definitions:

This section gives some of the definitions that are required for the explanation of the new protocol. In a distributed computing environment send, receive, internal computation and checkpoint at a process are events. The events are partially ordered by Lamport's [28] 'happens before (hb)' ($\rightarrow$) relationship. An event e_1 happens before e_2, if

(i) e_1 occurs before e_2 on the same process, or

(ii) e_1 is send of a message and e_2 is the receive of that message, or

(iii) $\exists\ e \mid e_1$ happens before e and e happens before e_2.

Any two events that cannot be ordered by *hb* relationship are *concurrent*.

Definition I: A *Global Checkpoint* (GC), denoted by $(C_{1,x},C_{2,y}$ ....,$C_{n,z})$ consists of one *local checkpoint* of each process.

Definition II: Checkpoint Interval: The i^{th} checkpoint interval at any process P_j denotes all the computation performed by P_j between i^{th} and $(i+1)^{th}$ checkpoint, excluding $(i+1)^{th}$ checkpoint.

Definition III: A *Consistent Global Checkpoint or State* (CGC/CGS) is a global checkpoint $(C_{1,x},C_{2,y}$ .. $C_{j,x,}$ $C_{k,y,}$ ..,$C_{n,z})$ such that:

Condition I: $\forall$m if receive(m) $\rightarrow C_{j,x} \Rightarrow$send(m) $\rightarrow C_{k,y}$; j,k $\in$ 1..n, j $\neq$ k;

Condition II: $\forall$m if $C_{j,x} \rightarrow$ receive(m) $\wedge$ send(m) $\rightarrow C_{k,y} \Rightarrow$m $\in$ stable; j,k $\in$ 1..n, j $\neq$ k;

The message *m* is stable when it is logged on to the stable storage.

Definition IV: Dependency Vector (dv): A vector of N fields at each process. $dv_i[i]$ is initialized to 1 and incremented on a checkpoint. It gives current interval index or the checkpoint index of the next checkpoint.

Definition V: Dependency Tracking: Each message sent by P_i is piggybacked with vector $dv_i[1..N]$. When P_j receives a message, P_j modifies its $dv_j[1..N]$ to be component wise maximum of dv_i and dv_j. This is used for forcing a checkpoint before receive a message in some quasi-synchronous checkpointing protocols if any field in $dv_i[1..N] > dv_j[1..N]$. This is known as *checkpointing condition*.

Definition VI: Dummy checkpoint: A dummy checkpoint of MH_i is a marker in the message logs of MH_i.

Definition VII: Actual Checkpoint: A checkpoint of MH_i that is taken and transferred to local MSS.

Definition VIII: Redundant Checkpoint: In quasi-synchronous checkpointing protocol, we define a redundant checkpoint to be one that is not forced due to the *checkpointing condition*.

3. Quasi-synchronous approach

Many quasi-synchronous checkpointing protocols have been proposed in the literature for distributed computing systems [1], [3], [4], [11], [22]. These protocols allow processes to take some of their checkpoints independently. Processes are also forced to take *forced checkpoints* to ensure domino-free recovery. These protocols fall in two categories: index based and model based. In index-based quasi-synchronous checkpointing protocols, each checkpoint is assigned a monotonically increasing integer called checkpoint sequence number (CSN). The checkpoints with same CSN form a consistent global state [8]. The protocols based on this approach piggyback CSN on each message. A process takes a checkpoint when it receives a message with higher CSN than its own. There are protocols that piggyback more information on messages to reduce the number of forced checkpoints but more piggybacked information does not always mean a reduction in the number of checkpoints [19]. Wang [20] provides a classification of the model based checkpointing protocols:

- force a *checkpoint after each send and before each receive* CASBR[20].
- force a checkpoint on each message receive CBR[20].
- force a *checkpoint after each send* CAS[20]
- force a checkpoint before receiving a message if receiver has sent one or more messages after its last checkpoint. This scheme is called no receive after send (NRAS) [1], [20], [22].
- force checkpoint before receiving a message that will change the dependency vector at receiver (FDI)
- force checkpoint before receiving a message that will change the dependency vector at receiver if receiver has sent one or more messages (FDAS).

These schemes require no extra checkpointing messages, no extra piggybacked information on each message (except FDI and FDAS) and are domino-free. But on the downside of it there are following overheads due to which they may not be very attractive for MDCSs:

- High frequency of checkpointing
- Complex recovery as the recovery protocol has to identify the set of local checkpoints that form a consistent global state
- Large stable storage requirement
- Purging Cost

The quasi-synchronous checkpointing protocols though have no synchronization overheads and do not block the underlying computation but may have high checkpointing overheads. For our protocol, we have chosen NRAS based approach, however the gains in performance are only indicative and can be achieved by using any other quasi-synchronous checkpointing scheme with our approach.

3.1 Analysis of NRAS Protocol

We first use a very general model to derive an expression for the average number of checkpoints taken by a process between two faults in NRAS scheme. The message-sending rate of a process (and therefore the message arrival rate at a process) is λ. When a process has to send a message it selects one out of the remaining process with equal probability. Let the fault arrival rate be ξ. Each process can be in one of the two modes: send and receive. Initially (and after recovery from a fault) each process is in receive mode. If a process is in receive mode and sends a message it goes to send mode. If a process is in send mode and receives a message it takes a checkpoint and goes to receive mode.

The state of a process at any time can be described by the two integer random variables, $I(t)$ and $J(t)$, where $I(t)$ is the state of the process (send or receive) and $J(t)$ is the number of checkpoints taken by the process . The process $X = \{[I(t), J(t)]; t \geq 0\}$, is an irreducible

Markov process with a state space $\{S,R\}\times\{0,1,\ldots\}$. The starting state and the state after each fault is $(R,0)$. The possible transitions of X are:

- (R,j) to (S,j) : with rate λ when the process sends a message in receive state
- (S,j) to $(R,j+1)$: with rate λ when the process receives a message in send state
- (x,j) to $(R,0)$: with rate ξ when a fault arrives where x can be R or S

We denote the joint steady state distribution of $I(t)$ and $J(t)$ by $\pi_{i,j}$:

$$\pi_{i,j} = \lim_{t\to\infty} P[I(t) = i, J(t) = j]; i = R,S; j = 0,1,... \quad (1)$$

The two-dimensional Markov process X can be easily reduced to a one-dimensional Markov process $Y = \{I(t); t \geq 0\}$ s.t. the state (i,j) of X is same as state $2j$ of Y (see Figure 1) if $i = R$ and $2j+1$ otherwise. Let us denote the steady state distribution of $I(t)$ by P_i:

$$P_i = \lim_{t\to\infty}\left[P(t) = i\right], i = 0,1,... \quad (2)$$

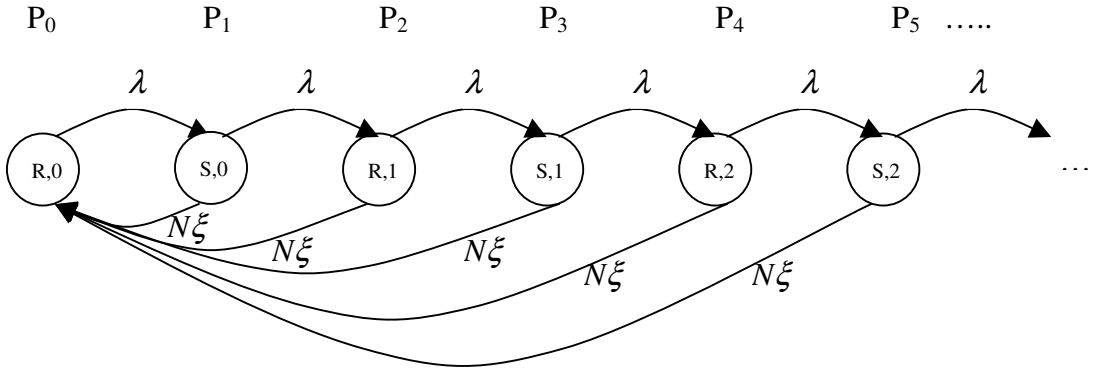


Figure 1: Markov Chain

These probabilities satisfy the following balance equations:

$$\lambda P_0 = N\xi(1-P_0) \quad (3)$$

$$\text{and } (\lambda + N\xi)P_i = \lambda P_{i-1}; \quad i = 1,2,\cdots \quad (4)$$

Equations (3) and (4) can be solved to get

$$P_0 = \frac{N\xi}{\lambda + N\xi} \quad (5)$$

$$P_i = \left(\frac{\lambda}{\lambda + N\xi}\right)^i P_0 \quad (6)$$

The average number of checkpoints N', taken by a process between two faults is given by:

$$N' = \sum_{i=2}^{\infty} \lfloor i/2 \rfloor (P_i) \quad (7)$$

Which can be solved to get:

$$N' = \frac{\lambda^2}{(\lambda + N\xi)^2 - \lambda^2} = \frac{\lambda^2}{N\xi(N\xi + 2\lambda)} \quad (8)$$

Equation (8) shows the relationship between the average number of checkpoints and message arrival rate in a NRAS based protocol. It can be seen that the **number of checkpoints increases almost linearly with increase in message arrival rate**.

Acharya and Badrinath [1] proposed a checkpointing protocol (AB protocol) for MDCS based on NRAS approach. A protocol designed to operate in a mobile environment requires proper handling of issues described in section 1. Their protocol stores the checkpoints of a MH on its current MSS. So, the checkpoints of MHs get distributed over the static network. This raises the problem of locating the checkpoints of MHs. The recovery protocol should not only identify the set of local checkpoints that forms a consistent global state but should also find out the location (i.e. MSSs) of the local checkpoints of MHs which are distributed over the static network. Acharya and Badrinath [1] gave an efficient solution for this using two vectors: dependency vector $CKPT_i[1\ldots N]$ and an associated *location vector* $LOC_i[1\ldots N]$. This scheme requires piggybacking of dependency vector and location vector along with each message and not only forces checkpoints when a MH receives a message after sending one but also forces a MH to take a checkpoint

- When the MH changes cell, or
- When the MH disconnects, or
- During recovery, if the required checkpoint is yet to be taken.

All this further increases the overhead of checkpointing, which is already very high in NRAS due to large number of forced checkpoints.

4. A Weighted Approach for Checkpointing

4.1 Basic Idea

In our scheme, a MH does not always take a checkpoint when a checkpoint condition occurs. It skips the checkpoint if the probability of using it during the recovery is very low. This reduces the load on wireless channels, as every checkpoint taken by the MH need to be stored on its current MSS. If a skipped checkpoint is required for recovery, the recovery procedure constructs it using the message logs and the last actual checkpoint.

Our scheme is motivated by the fact that *the messages, which a MH receives, are routed through the current MSS of the MH*. So, the messages to a MH can be logged at MSS. The AB protocol also logs messages to handle in-transit messages. But our protocol uses these message logs to reduce the load on wireless links due to checkpointing. Moreover, no extra checkpoints are needed on disconnection. The basic idea behind the protocol can be described as follows:

- All static nodes follow strict NRAS quasi-synchronous approach to take checkpoints.
- All MHs use NRAS approach to take checkpoints, but actually record only those checkpoints that may be needed during recovery with high probability. A checkpoint, which may be needed during recovery with a very low probability, is recorded as a dummy checkpoint. MSSs log the messages received for the MHs. If a dummy checkpoint is needed during recovery it is constructed from previous checkpoint and logs.

The message logs allow us to handle the faults at MH (which are quite frequent) in a different way (and with less overheads) than the faults at MSS (which are quite infrequent). Recovery from a fault at MH does not involve any other node whereas a fault at MSS requires a global recovery. The recovery in the two cases (MH / MSS) is explained below.

Asynchronous Recovery of MHs

Our approach allows MHs to recover independently. The message logs and the checkpoints of the MHs are distributed over the static network. These need to be organized efficiently for fast recovery of MHs. The simple scheme that we propose is based on a data

structure MSS_list_i associated with each MH_i. The data structure MSS_list_i records the movement of MH_i and is transferred to new MSS_s during handoff. It is also stored as the part of a dummy checkpoint. A new MSS_x adds itself to MSS_list_i only when it logs first message for the MH_i in its cell. Thus, all those MSSs visited by MH_i where no logs are stored for MH_i are not part of this data structure.

MSS_list_i is available at the current MSS of the MH and gives the list of all MSSs visited by MH since last actual checkpoint where the logs and actual checkpoint for the MH are stored. The first MSS is the one that has the last actual checkpoint of the MH. During recovery a message is sent to all MSSs in the list MSS_list_i for actual checkpoint and message logs of the MH. On receiving this message all MSSs send the actual checkpoint and message logs to the current MSS. The current MSS sends the actual checkpoint to the MH and replays the messages from the log. At the completion of replay of the logs, the recovery of MH is over.

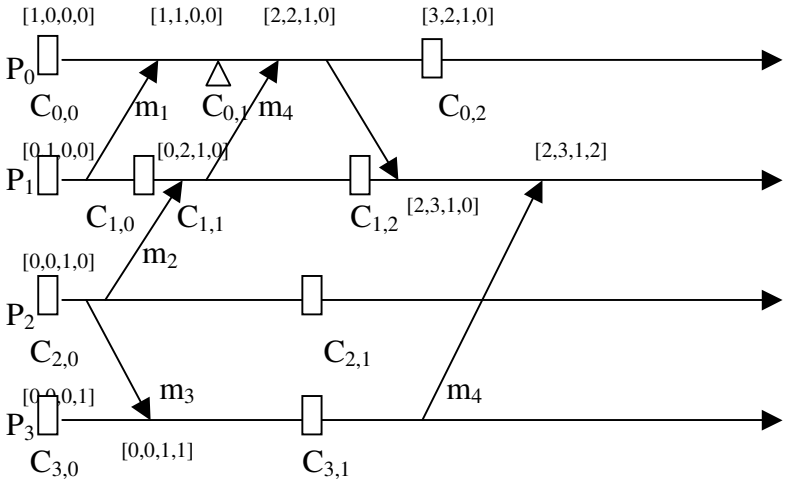
Recovery due to the fault at MSS

A fault at MSS invokes global recovery. Here, a dependency vector similar to AB protocol [1] is used to identify recovery line. To identify location of checkpoints, we propose to use *location directories (LDs)* maintained by different MSSs instead of using location vector. Each MSS has many checkpoints of different visitor MHs. Each MSS_s stores the *process_id* and *CSN* of checkpoints of MHs in its location directory. If MH_i takes checkpoint p when it was local to MSS_s, MSS_s location directory will have an entry (i, p) in it. During global recovery, the dependency vector is sent to all MSSs. Each MSS searches its location directory for the required checkpoints and responds if found. The impact of this change is that MHs need not checkpoint on location change. This **also reduces the piggybacked information** with each message.

We now discuss some more observations that motivated the design of our protocol. When process P_j sends message *m* to process P_i, it piggybacks the sequence number of the next checkpoint to be taken by P_j in the j^{th} field of dependency vector sent with *m*. If P_i takes a checkpoint on delivery of message *m* then this dependency is recorded in the dependency vector stored with the next checkpoint of P_i. This dependency can also be created transitively (P_j sends m_1 to P_k and P_k sends m_2 to P_i). Thus, more is the number of messages sent in checkpoint interval *j* more is the possibility that the checkpoint *j* is recorded as part of the global consistent state in the dependency vector of other processes and more is the probability that this checkpoint is needed during recovery. Consider Figure 2, process P_2 sends two messages in first checkpoint interval and thus the checkpoint $C_{2,1}$ is part of consistent global state at processes P_0, $P_1$and P_3.

Observation II: More is the number of sends in k^{th} checkpoint interval of P_i more is the probability that k^{th} checkpoint of P_i will be needed during recovery. **So, such checkpoints must not be skipped by the MHs**.

In this Figure, all processes take a checkpoint at the start and initialize their dependency vectors to all zeroes except for the entry for itself that is initialized to one.



Process P_0 has not sent any message in this checkpoint interval but takes a checkpoint $C_{0,1}$ due to location change. The checkpoint $C_{0,1}$ is redundant as the checkpoint $C_{0,0}$ can be used in place of it.

Observation III: If a process does not send a message in a checkpoint interval and takes a checkpoint due to location change or disconnection, then this checkpoint is a *redundant checkpoint*.

4.1 Weighted Checkpointing Protocol

The above example shows that the checkpoints taken by a process in AB approach can be *redundant* or *useful* with respect to their inclusion in the global consistent state at the time of recovery. Our idea is to eliminate all *redundant* checkpoints of AB approach and to take *useful* checkpoints at MHs *selectively*. The proposed protocol assigns a weight to each checkpoint at the MHs. This weight reflects the ability of a checkpoint to be the part of global consistent state. If the weight associated with a checkpoint exceeds a limit, the checkpoint must be taken. We consider following factors for assigning weights to the checkpoints:

1) number of messages *sent* since last checkpoint,
2) number of checkpoints skipped, and
3) number of location changes since last checkpoint

First factor is due to *observation II*. More is the number of sends before a checkpoint more processes will record the checkpoint in their respective recovery lines and more will be probability that the checkpoint is required during recovery. The second factor affects the cost of reconstruction of a checkpoint. More is the number of consecutively skipped checkpoints more is the probability that a dummy checkpoint is required at the time of recovery. Also, the number of messages received from other processes increase with increase in time since last actual checkpoint, which increases the cost of reconstruction of a skipped checkpoint if required at the time of recovery. The third factor gives an approximate measure of the number of MSSs that need to be contacted for construction of a skipped checkpoint at the

time of recovery. More is the number of MSSs that need to be contacted to construct a skipped checkpoint more efforts are required for the construction of checkpoint and slower will be recovery. The fault rate can also be considered as a factor. More is the fault rate more frequent is the recovery and more is the probability that a checkpoint is required for recovery.

We assign a weight to each of the above events in the MDCS (see section 5.1). Initially the weight associated with the process is zero. The weight of the process is increased by the weight of the event, when it occurs. When a NRAS event takes place, the weight of the process is compared with the predefined threshold value and a checkpoint is taken only if the weight of the process is greater than or equal to the threshold value. The weight is initialized to zero after a checkpoint. This scheme is only implemented for MHs and leads to *great reduction* in checkpoints taken by MHs.

Assumptions

- All checkpoints taken by a MSS/ MH along with dummy checkpoints are numbered in strictly increasing order.
- All processes are assumed to exhibit piece-wise deterministic behavior.
- We assume that each MH or MSS executes only one process at a time. This assumption can be relaxed easily.

Algorithm

The Weighted No Receive After Send (WNRAS) algorithm is explained here. MSSs and MHs executes same algorithm leaving the parts of the algorithm for location management, disconnection and skipping of checkpoints, which is executed by MHs only. The data structures used in the algorithm are as follows:

P_i: represents i^{th} process running on either MH or MSS;

d_vector[]: is the dependency vector with each process, $d\_vector_i[i]$ initialized to one and all other fields are zero initially; $d\_vector_i[i]$ is incremented when P_i takes a checkpoint and d_vector[] is updated on message receive;

m_d_vector[]: is the dependency vector piggybacked on message *m*;

MSS_list_i: list of MSSs having message logs for MH_i since last actual checkpoint, stored at current MSS of the MH;

wt_i: weight associated with next checkpoint of P_i; initialized to zero at each checkpoint;

w_{send}: P_i increments wt_i by w_{send} when it sends a message;

w_{skip}: P_i increments wt_i by w_{skip} when it skips a checkpoint;

w_{dlc}: P_i increments wt_i by w_{dlc} when it changes a cell or disconnects;

T: threshold value, P_i takes its next checkpoint if $wt_i \geq T$

l_dir_i: is location directory at MSS_i and contains list of MHs and their checkpoints with MSS_i;

mode: of any node can be either *send* or *receive*;

CP_ln: identity of MSS_s of MH_i before disconnection.

first(m): a function that returns true if m is the first message forwarded for a MH in the MSS cell;
recovery_flag$_i$: flag associated with MH_i, set to one if there was a recovery during MH_i's disconnection;
n: total number of MHs;
m: total number of MSSs;

P_i sends a message to P_j

```
if (mode[Pi] = = receive){ mode[Pi] = send;}
append "d_vector" with the message;
if (Pi is MH)  wti =wti + wsend;
send message;
```

P_i receives message m from P_j

```
/* executed by MSSs for local MHs */
if((Pi = = MSS)&&(m for MH)){
  log m;
  forward m to MH;
  if (first(m))
    add MSS to MSS_listi; exit;}      /* end if */
if (mode(Pi) = = receive) {           /* if process is in receive mode, just receive the
  receive message;                       message */
  for(i=1; i< =n; i++)
  if(m_d_vector[i] > d_vector[i])
    d_vector[i] = m_d_vector[i];      /* update dependency vector */
  exit;}                              /* end if */
   /* if process is in send mode */
if (Pi = = MSS){                      /* if process is on MSS, take checkpoint */
  checkpoint( );}
else{                                 /* if process is on some MHi connected to MSSs */
  if (wti > T){                       /* if process wti is greater than T */
    checkpoint( );
    transfer checkpoint to MSSs;
    wti =0; }
  else{
    skip checkpoint;
    wti =wti + wskip;
    send dummy checkpoint message to MSSs;
    }/* end else */
  increment d_vectori[i] by one;
  mode[Pi] = receive;
  receive message;
  for(i=1; i< =n; i++){
   if(m_d_vector[i] > d_vector[i]){
     d_vector[i] = m_d_vector[i];}} /* update dependency */
  };
checkpoint( ) /* procedure to take a checkpoint of process Pi*/
  {
  if (MSS)   transfer the state of the application and d_vector to stable storage;
  if (MH)    transfer the state of the application and d_vector to local MSS;
  }
```

MSS_s receives MH_i checkpoint q or dummy checkpoint message
transfer MSS_list_i to stable storage;
update l_dir_s; /* MSS_s enters MH_i and checkpoint index *q* in l_dir_s */
if (q is actual){
  transfer checkpoint to stable storage;
  reinitialize mss_list_i;}
else
  mark dummy in message log_i;

MHs disconnects from MSS_r
  send MSS_r disconnection message;
  $wt_i = wt_i + w_{dlc}$;
  $CP\_ln \leftarrow MSS_r$;
  disconnect;
MH_i changes cell of MSS_s to MSS_r
  $wt_i = wt_i + w_{dlc}$; /* MH_i increase its wt */
  MSS_s transfer mss_list_i to MSS_r; /* handoff */
  update mss_list_i; /* MSS_r updates the mss_list_i */

MH_i reconnects to MSS_s
  if(**valid**(CP_ln){
    old_mss ← CP_ln; /* if MH_i has identity of MSS before disconnection */
  else old_mss ← NULL;
  send MSS_s reconnect(old_mss);
  get reconnect information or checkpoint from MSS_s;
  rollback if required; /* if rollback message with checkpoint is received then rollback */
  connect;
MSS_s receives reconnect(old_mss) message from MH_i
  if(old_mss = =NULL) **discover** MSS_r; /* if MH is not having identity of old_mss, find it */
  get information from MSS_r for MH_i;
  if($recovery\_flag_i$){ /* if recovery occurred during disconnection */
   send rollback message and checkpoint to MH_i; ;}/* send rollback checkpoint to MH_i */
  else transfer reconnect information to MH_i;
  end.

4.2 Purging

The purging of the distributed checkpoints and message logs is very important as the stable storage of the MSS is limited. The information that is not required in future for recovery should be discarded. All MSSs cooperate to purge the obsolete information. Any MSS can initiate the purging algorithm. The initiating MSS sends request to each MSS to send the dependency information to it. When a MSS receives the request for dependency information, it takes component wise minimum of its own dependency vector with dependency vectors of all connected and disconnected MH's with it and sends it to the

initiator. The initiator on receiving all dependency vectors takes component wise minimum of its own dependency vector with the received dependency vectors and sends it to all MSSs. All MSSs then discard those entries, which are less than these minimum values. In case of dummy checkpoints, the previous actual checkpoint and message logs are retained.

4.3 Recovery for Weighted Checkpointing

Here, we describe the recovery protocol for the above checkpointing protocol. Let there be a fault at process P_i running on a MH or MSS. The recovery due to a fault at MH is different than the recovery of MSS.

4.3.1 Fault at MH

In this case, the last actual checkpoint $C_{i,n}$ taken by process P_i and the message logs are sufficient to restore MH's state just before the fault. On recovering from the fault, MH_i sends a request to its local MSS_s for recovery. The MH_i piggybacks the address of last MSS (say MSS_t) before the fault with the request if it remembers. On receiving the request, the current MSS_s discovers the MSS_t if $t \neq s$. If MH_i does not remember last MSS then MSS_s uses a broadcast containing the address of MH_i to discover MSS_t. If MSS_s and MSS_t are same, MSS_s contains the MSS_list_i for the MH_i, otherwise, MSS_s gets the MSS_list_i for MH_i from the MSS_t. MSS_list_i contains the address of last MSS that has the actual/dummy checkpoint of MH_i and the addresses of all MSSs where message logs after the actual / dummy checkpoints are stored.

Then MSS_s sends the request to MSSs in MSS_list_i for actual checkpoint $C_{i,n}$ and message logs of MH_i. After receiving the checkpoint and message logs MSS_s loads the checkpoint $C_{i,n}$ and replays the log of messages to MH_i. As the recovery at a MH does not involve other MHs, our algorithm can tolerate *any number of MH's failure simultaneously*.

The data structures and the algorithm used for recovery from a fault at MH are as under:

local_MSS: is the current MSS of MH_i;

send (recovery_request, local_MSS, old_mss): MH_i sends message to local_MSS to recover with identity of MSS (CP_ln) it was connected with before fault if it remembers otherwise sends NULL;

MSS_list_i: A list for MH_i that has the MSSs visited since last checkpoint; if it is {MSS_1, MSS_2,…,MSS_k}, then MSS_1 has actual checkpoint and others have logs of MH_i;

l_dir_x: represents the location directory at MSS_x;

Recovery from fault at MH_i connected to local MSS_s

```
recovery_mh(){
  if(valid(CP_ln){
    old_mss ← CP_ln;        /* if MH_i has identity of MSS_r it was connected to before fault */
  else  old_mss ← NULL;
  send (recovery_request, MSS_s, old_mss);
  receive recovery checkpoint and message logs from {MSS_1, MSS_2,…,MSS_k};
  restore checkpoint C_i,x to MH_i;
  replay messages in the log;
  send recovery over message;
```

MSS_s receives recovery request from local MH_i

```
  if(old_mss = =NULL) discover MSS_r;  /* if MH is not having identity of old_mss, find it */
  get MSS_list_i from MSS_r for MH_i;
  find last actual checkpoint C_i,q from MSS_list_i;
  find location of actual checkpoint and logs from MSS_list_i;
  send request for checkpoint/message logs of MH_i to MSSs in {MSS_1, MSS_2,…,MSS_k};
  receive checkpoint and/or message logs from {MSS_1, MSS_2,…,MSS_k};
  load checkpoint to MH_i;
  replay log;
   }
```

4.3.2 Fault at MSS

The recovery due to the fault at MSS is global and requires coordination between all processes. The complete recovery line (CGS) of the distributed mobile system is stored along with the checkpoint as the dependency vector. Let, process P_i on MSS_i fails. On recovering from the fault it rolls back to its last checkpoint $C_{i,n}$. Let, $(C_{0,x},C_{1,y}$ ... $C_{i,n},...,C_{n,z})$ be the dependency vector associated with $C_{i,n}$. This dependency vector gives the consistent global state of the MDCS to which all processes including the processes on MHs must roll back.

The global recovery procedure needs to do some or all of the following:

- locate checkpoints of MHs that are part of recovery line
- construct dummy checkpoints that are part of recovery line
- recovery of disconnected MHs
- construct the checkpoint of the disconnected MHs that are part of recovery line

- transfer checkpoints to the respective MHs asking them to roll back

There are two cases when an actual checkpoint may not be available.

- MH or MSS has not taken the checkpoint yet
- It is a dummy checkpoint

In first case, the checkpoint is forced to the MH or MSS and then response is sent to the coordinator. This is simple in case of MSSs or connected MHs because a checkpoint can be forced to them easily. In case of disconnected MHs the situation is critical if a checkpoint required for recovery is yet to be taken. This checkpoint needs to be constructed from the last actual checkpoint and the message logs of MH similar to a dummy checkpoint.

In second case, the MSS that has the entry for the dummy checkpoint in its location directory constructs the checkpoint of the MH_i required for the recovery. The simple procedure is: retrieve the MSS_list_i for MH_i from dummy checkpoint. Send messages to the MSSs in MSS_list_i for actual checkpoint and logs of MH_i. Receive the checkpoint and logs of MH_i up to the dummy checkpoint. Then, the MSS creates a process and loads the actual checkpoint of the MH_i. The messages from the logs are replayed. After the logs are over, the process reaches a state where the dummy checkpoint was marked. The resulting state of the process is checkpointed which is the required dummy checkpoint of MH_i.

Here, we explain the procedure of constructing a dummy checkpoint with an example. In Figure 3, let MH_x has its actual checkpoint $C_{x,n-1}$ on MSS_1 before a dummy checkpoint $C_{x,n}$ at MSS_n that is needed for the recovery. MH_x visits the cells of MSS_1 to MSS_n through MSS_2, MSS_3 etc. after taking the actual checkpoint $C_{x,n-1}$ and before taking dummy checkpoint $C_{x,n}$. During recovery, recovery coordinator sends the complete recovery line to all MSSs. MSS_n finds out the entry of dummy checkpoint $C_{x,n}$ in its l_dir_n. To construct the dummy checkpoint $C_{x,n}$ of MH_x, MSS_n checks the MSS_list_x with it and finds out that the last actual

checkpoint $C_{x,n-1}$ is at MSS_1. MSS_n sends request for checkpoint $C_{x,n-1}$ to MSS_1 and in response gets the actual checkpoint $C_{x,n-1}$ and message m_a as log of MH_x at MSS_1.

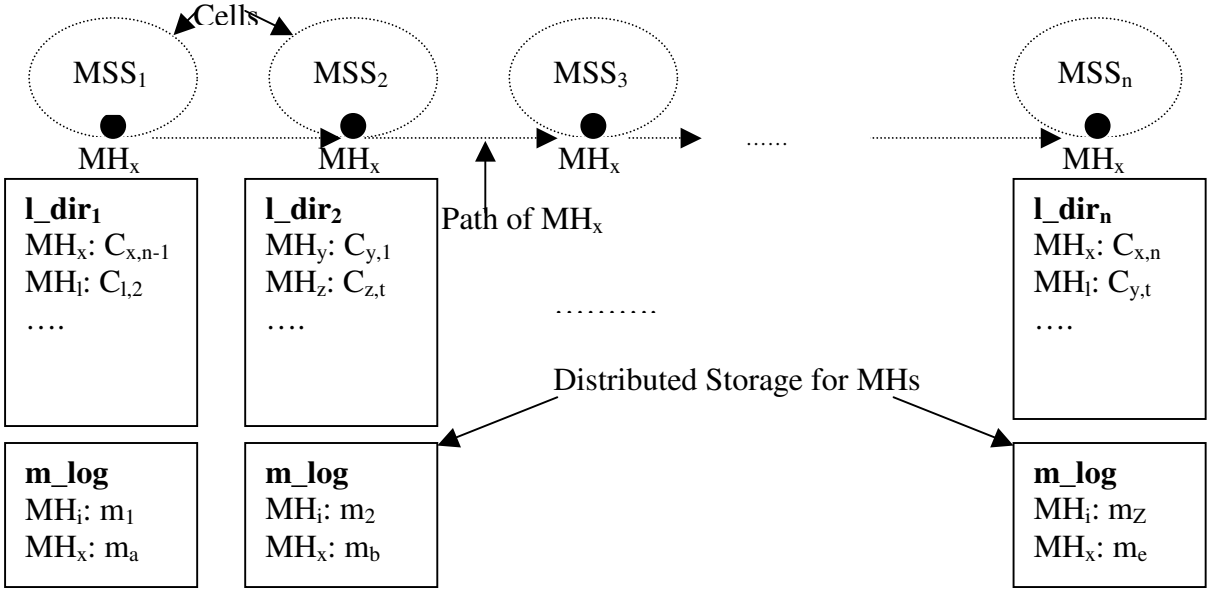


Figure 3: Distributed Stable Storage of Mobile Host

Simultaneously it sends request for message logs to MSS_2, MSS_3 and so on up to MSS_{n-1} in MSS_list_i. After receiving the $C_{x,n-1}$ from MSS_1 and message logs from all other MSSs (m_a, m_b etc.in Figure 3), MSS_n checks if it has logged messages for MH_x before $C_{x,n}$ (m_e in Figure 3). The MSS_n creates a process with initial state as $C_{x,n-1}$ and replays logs (m_a, m_b, …,m_e). The state of the process is checkpointed as $C_{x,n}$ and then MSS_n responds to the recovery coordinator on behalf of MH_x.

Similar is the procedure for creating a checkpoint for the disconnected MH if it has not taken the required checkpoint before disconnection. When the message containing recovery line from the recovery coordinator reaches any MSS_k, it checks disconnected_MH list with it. Then it checks if the index of the checkpoint of any disconnected MH_x that is part of recovery line is greater than the index of checkpoint of MH_x available with it in MSS_list_x. If so, it needs to construct the disconnected checkpoint of MH from the last actual checkpoint and

message logs. The procedure given above to construct the dummy checkpoint is repeated. The disconnected MH rolls back to this checkpoint on reconnection.

Here, we give the algorithm to construct a dummy checkpoint or a disconnected checkpoint. The procedure is executed by say MSS_k to construct a dummy/disconnected checkpoint $C_{x,n}$ from actual checkpoint $C_{x,n-1}$ at MSS_1 and message logs at MSS_1,MSS_2,….,MSS_k.

construct_dummy(){
receives recovery line ($C_{0,x}$,$C_{1,y}$ ... $C_{x,n}$,...,$C_{n,z}$);
if(dummy($C_{x,n}$)){
for(all MSSs in MSS_list_x){/* MSS_list_x is MSS_1,MSS_2,….,MSS_k traversed by MH_x*/
send request to MSSs in MSS_list_x for checkpoint/message logs of MH_x;}
receive checkpoint $C_{x,n-1}$ and message logs of MH_x;
create a process for MH_x and load the checkpoint $C_{x,n-1}$;
replay the message logs to the process;
if(all messages replayed)
checkpoint the state of process;
save it on stable storage;
change status of checkpoint to actual;
}
In our scheme the overhead of taking a checkpoint at every disconnection (assumed by most of the checkpointing algorithms [1], [5], [6] for mobile distributed systems) is saved.

Now, we present the complete recovery protocol for the fault at MSS. Let us assume that the fault is at some MSS_r and the recovery line stored at the last checkpoint $C_{r,t}$ of MSS_r is ($C_{0,x}$,$C_{1,y}$ ... $C_{r,t}$,...,$C_{n,z}$). The recovery protocol is as described below.

Recovery from fault at MSS
$last\_check_i$: last checkpoint of MH_i before disconnection;
$recovery\_flag_i$: A flag at a MSS_s for each MH_i that disconnected from MSS_s and indicates if the MH_i need to rollback on reconnection or not;
$disconnected\_MH_s$: list of MHs that disconnected from MSS_s and has not connected to any MSS so far;
$active\_MH_s$: list of MHs that are currently in the cell of MSS_s;
recovery_mss(){ /*MSS_r */
{
rollback to last checkpoint $C_{r,t}$;
/* let ($C_{0,x}$,$C_{1,y}$ ... $C_{r,t}$,...,$C_{n,z}$) be the d_vector associated to checkpoint $C_{r,t}$ */
for(i =1; i<=m; i++){ /* To all MSSs send recovery line */
send recovery message with ($C_{0,x}$,$C_{1,y}$ ... $C_{i,n}$,...,$C_{n,z}$);}
receive responses from the MSSs; /* MSSs sends own responses and responses on behalf of connected and disconnected MHs */

```
  if (responses received from all processes){
    send commit to all processes;}
```

MSS$_p$ recives recovery message

```
 send recovery message to all attached MHs;
 if(d_vector_p[p] > C_p,l ){     /* C_p,l is checkpoint of MSS_p required for recovery */
   send coordinator response;}
 else{
  take checkpoint C_p,l;
  send response to the coordinator; }
for(i =1; i<=n; i++){      /* n is total number of MHs */
  if(recovery_line[i] is in l_dir_p){/* if checkpoint of MH_i required for recovery is at MSS_p */
   if(dummy(recovery_line[i])){/* if MH_i's checkpoint required for recovery is dummy */
    construct_dummy()}
   send response to MSS_r on behalf of MH_i;}
for(i =1; i<=n; i++){
 if(MH_i ∈ disconnected_MH_p){
  if(recovery_line[i] > last_check_i){
   construct_dummy()} /* construct the disconnected checkpoint for the MH_i */
 send response to MSS_r on behalf of MH_i;}
```

5. Performance Evaluation

5.1 Simulation

For simulation we consider a MDCS with 16 MHs and 4 MSSs. Each node generates messages using exponential distribution with mean λ. The destination of the message is selected randomly with equal probabilities. The message communication delay has exponential distribution with mean equal to β. Arrival of faults is assumed to be Poisson with rate ξ. We first study the effect of change in message arrival rate and try to assign weights to different events described in section 3.2. We then compare our algorithm with the algorithm proposed by Acharya and Badrinath [1] for different values of λ, β, and ξ. The comparison is based on the time needed to process N messages i.e., 10,000. Each MH remains in a cell with exponentially distributed time with mean x. A MH may disconnect from the distributed system with 50% probability and with 50% probability it may switch to the other cell. When a MH disconnects, it reconnects itself to the distributed system after an exponentially distributed time with mean y. We assumed x =500 seconds and y = 100 seconds.

5.1.1 Weight Assignment: Weight assignment to different events is done using heuristics. Each process maintains a variable wt_i initialized to zero. It is incremented by an event weight when an event occurs. The weights associated with the events are assigned by repetitive trials. For each trial we fix two out of three weights and vary the third weight. The best value is found for the varied weight. This tentative best is used to fix another weight and so on. This process is repeated until there is no significant variation. When a process has to take a checkpoint it compares it's own weight (wt_i) with a threshold value. Also, we used following two parameters as our performance metric.

d_r = Total number of dummy checkpoints used for recovery / Total number of actual checkpoints required for recovery.

d_t = Total number of dummy checkpoints / Total number of actual checkpoints taken.

The first ratio gives a measure of recovery cost and the latter gives the measure of gain due to the dummy checkpoints. Our objective is to reduce the recovery cost due to dummy checkpoints and at the same time increase the reduction in checkpointing cost. After fixing these weights, we repeat the simulation until there is no further change in d_r and d_t. Final values assigned to w_{skip}, w_{send} and w_{dlc} are 0.08, 0.26 and 0.43 respectively.

Discussions:

The simulation results assign weights to various events identified earlier. We found that these values give considerable improvement in the performance of the protocol. The weights assigned are tunable and can be changed. This makes the protocol adaptable. If the weights assigned are too less, most of the checkpoints will be skipped and protocol will behave as log based protocol and if the weights assigned to the events are high, the protocol will behave as NRAS protocol.

5.1.2 Comparison of Results

Figure 4 shows the average number of checkpoints taken between two faults in case of AB and our approach as a function of sending rate. As expected the number of checkpoints in our approach is far less than AB approach.

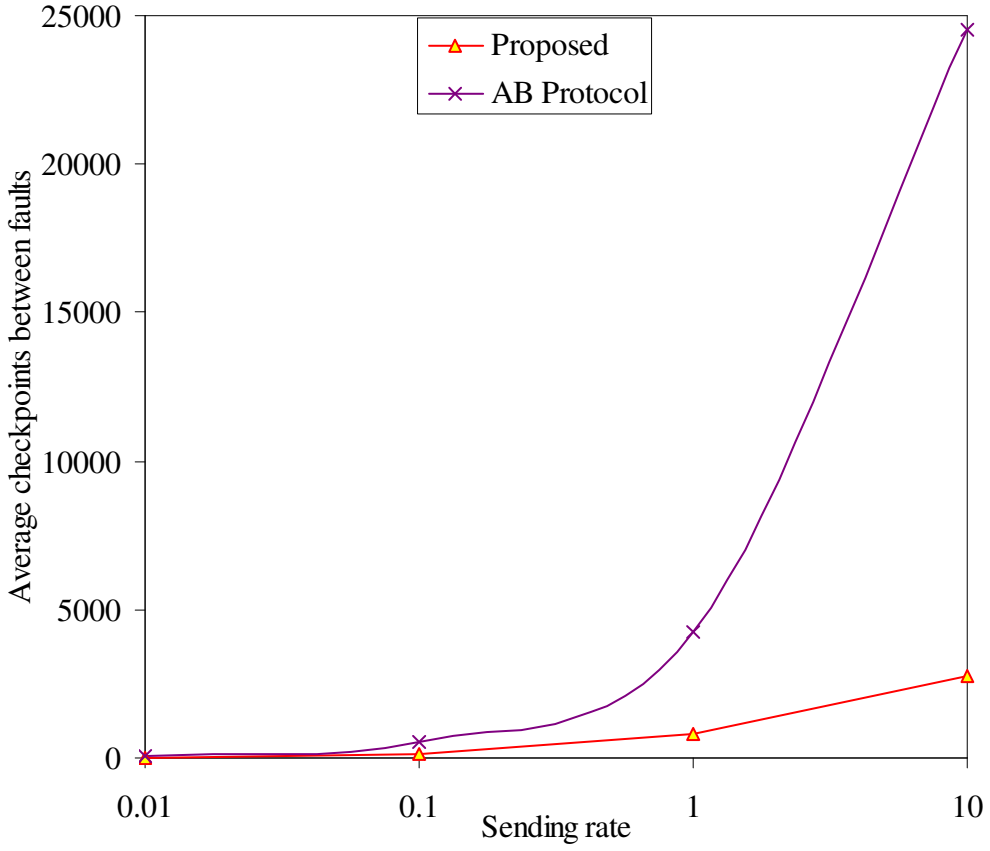


Figure 4: Average number of checkpoints in AB and our approach as function of sending rate for λ=1, ξ=.0001.

Discussions: The number of checkpoints in AB approach increases rapidly with increase in sending rate, whereas in our case the increase is gradual. For any checkpointing protocol for MDCSs, this is highly desirable, as it will ensure constant load on wireless network. At higher sending rates, number of checkpoints in AB approach is unpredictably high.

Next, we examine the impact of dummy checkpoints during recovery. Figure 5 shows an interesting phenomenon. With increase in sending rate, ratio d_t increases and ratio d_r decreases. This means that with increase in the sending rate there is increase in the checkpoints that are skipped and also reduction in the number of dummy checkpoints required for recovery, i.e., cost of constructing dummy checkpoints for recovery.

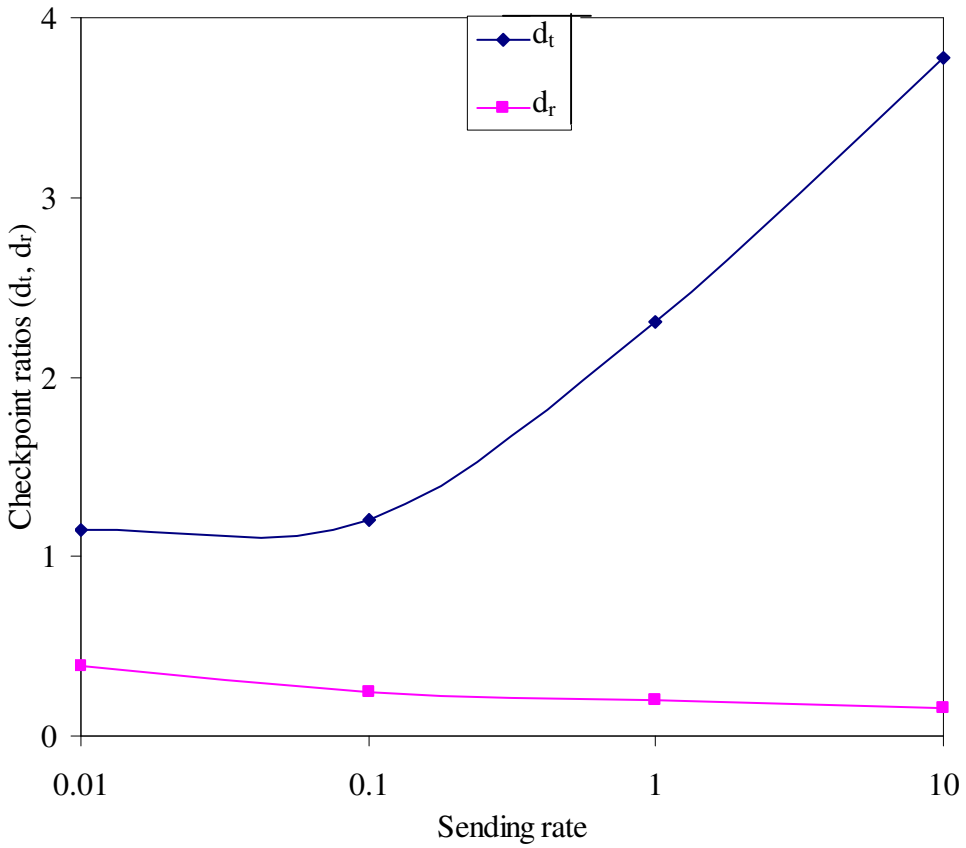


Figure 5: Checkpoint ratios in our approach as function of sending rate for λ=1, ξ=.0001.

Discussion: Thus, the performance of our protocol improves with increase in message-sending rate. The number of dummy checkpoints required for recovery is not more than 20% for various sending rates. When sending rate is one, the dummy checkpoints are only 15% of the total number of checkpoints required for recovery (approx. three out of 20 checkpoints) whereas the total number of skipped checkpoints is more than double the actual checkpoints taken. Thus, the hybrid checkpointing has very less overhead of reconstructing checkpoints from message logs compared to huge savings due to the skipped checkpoints.

5.1.3 Comparison

The proposed protocol is different than the coordinated checkpointing protocols proposed for MDCSs. Here, we have compared this protocol with the coordinated protocol proposed by Cao-Singhal [6]. The CS [6] protocol has almost constant checkpointing overhead throughout whereas in proposed protocol the checkpointing overhead increases gradually. So, there will be a point where both these overheads will cross. Below that point the proposed protocol will be better and above that CS protocol will be better on checkpointing overhead.

Next, in the worst case, CS protocol requires 3*N messages for each consistent checkpoint whereas there is no message overhead for our protocol. But, in our protocol there will be overhead of purging messages periodically on wired network only.

Next is the piggybacked information on each message. In CS approach, the messages for checkpointing from initiator carry its address, MR, recv_csn, msg_trigger, req_csn and recv_wt etc. In the proposed protocol each message contains the dependency vector.

During recovery, in CS approach the checkpoints of all the MHs may have to be transferred to current MSS and then to the MH. In proposed protocol for the fault at MH only few MSSs need to be contacted and other MHs or MSSs are not disturbed. In case of a fault at MSS (which are rare), a global recovery is executed that require transfer of checkpoints and logs for few MHs. In this case, due to the replay of logs the recovery mayl be slower than recovery in CS approach.

6. Discussion of Results and Conclusions

In this paper we propose an adaptive approach for checkpointing MDCSs. This approach links the recording of a checkpoint at mobile nodes with sending of messages, skipping of checkpoints and disconnection or location change of mobile nodes. The skipped checkpoints of MHs, are constructed from the message logs and actual checkpoints at the MSSs. Thus our approach gives a relief to MHs from the heavy load of checkpointing. The proposed protocol decreases the total number of checkpoints taken by MHs, load on wireless channels and adds only few extra messages on wireless channels during recovery. The faults at MH and MSS are treated differently. The fault at MH (which are frequent) has less cost as compared to fault at MSS.

We compare our approach with Acharya and Badrinath [1]. In a highly mobile system, their protocol has high checkpointing overheads on MHs and thus high checkpoint transfer load over mobile network. Our approach reduces these overheads significantly. The performance of our protocol increases with increase in sending rate as the number of dummy

checkpoints increase which reduces the checkpointing overheads and further the dummy checkpoints required for recovery decrease which decreases the overheads during recovery.

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] Chi-Yi Lin , Szu-Chi Wang , Sy-Yen Kuo, "An efficient time-based checkpointing protocol for mobile computing systems over mobile IP", Mobile Networks and Applications, v.8 n.6, p.687-697, December 2003.

[3] 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.

[4] Briatico D., Ciuffoletti A. and Simoncini L., "A Distributed Domino-Effect Free Recovery Algorithm," Proceedings of International Symposium on Reliable Distributed Software and database, pp. 207-215, December 1984.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[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] Forman G.H. and Zahorjan J., "The Challenges of Mobile Computing," Computer, vol. 27, no. 4, pp. 38-47, April 1994.

[11] 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.

[12] 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.

[13] 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.

[14] Neves N. and Fuchs W. K., "Adaptive Recovery for Mobile Environments," Communications of the ACM, vol. 40, no. 1, pp. 68-74, January 1997.

[15] Park T., Woo N., Yeom H. Y., "An Efficient Recovery Scheme for Mobile Computing Environment," International Conference on Parallel And Distributed Systems (ICPADS), pp. 53-60, June 2001.

[16] Pradhan D.K., Krishana P.P. and Vaidya N.H., "Recovery in Mobile Wireless Environment: Design and Trade-off Analysis," Proceedings 26th International Symposium on Fault-Tolerant Computing, pp. 16-25, 1996.

[17] 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.

[18] 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.

[19] Tsai J., Kuo S.Y. and Wang Y.M., "Theoretical Analysis for Communication-induced Checkpointing Protocols with Rollback-dependency Trackability," IEEE Transactions on Parallel & Distributed Systems, vol. 9, no. 10, pp. 963-971, October 1998.

[20] Wang Y. M., "Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints," IEEE Transactions on Computers, vol. 46, no. 4, pp. 456-458, April 1997.

[21] Yao B., Ssu K. F. and Fuchs W. K., "Message Logging in Mobile Computing," Proceedings of the 29th Annual IEEE International Fault-Tolerant Computing Symposium, pp. 294-301, June 1999.

[22] Russel D. R., "State restoration in Systems of Communicating Processes," IEEE Trans. Software Engineering, vol. 6, no. 2, pp. 183-194, March 1980.

[23] Park T., Woo N., Yeom H. Y., "An Efficient Optimistic Message Logging Scheme for Recoverable Mobile Computing Systems" IEEE Trans. Mobile computing, vol. 1, no. 4, pp. 265-277, December 2002.

[24] JinHo Ahn , Sung-Gi Min , Chong-Sun Hwang, "A causal message logging protocol for mobile nodes in mobile computing systems", Future Generation Computer Systems, v.20 n.4, p.663-686, May 2004

[25] Vaidya N. H., "Staggered Consistent Checkpointing," IEEE Transaction Parallel and Distributed Systems, vol. 10, no. 7, pp.694-702, July 1999.

[26] Lalit Kumar, Manoj Misra, Joshi R. C., Checkpointing and Recovery for Distributed Computing Systems. In P. Ezhilchelvan and A. Romanovsky (eds.), Concurrency in Dependable Computing, Kluwer Academic Publishers, pp. 273-288, 2002.

[27] Lalit Kumar, Manoj Misra, Joshi R. C., "Low overhead Optimal Checkpointing for Mobile Distributed Systems", Proceeding IEEE ICDE 2003, pp. 686-688, March 2003.

[28] Rao S., Alvisi L., and Vin H. M., "The cost of recovery in Message Logging Protocols," IEEE Transaction Knowledge and Data Engineering, vol. 12, no. 2, pp.160-173, March/April 2000.

[29] Lamport L., "Time, Clocks and Ordering of Events in Distributed Systems," Comm. of the ACM, July 1978.

[30] Guo-Hui Li, Hong-Ya Wang, "A novel min-process checkpointing scheme for mobile computing systems", Journal of systems Architecture: the EUROMICRO Journal, v. 51, n. 1 pp. 45-61, January 2005.