A Tunable Checkpointing Algorithm for the Distributed Mobile Environment

Sungchae Lim

Dept. of Computer Science, Dongduk Women's University
Seoul, 136-714, South Korea

Abstract

The aim of a distributed checkpointing algorithm is to efficiently restore the execution state of distributed applications in face of hardware or software failures. Originally, such algorithms were devised for fixed networking systems, of which computing components communicate with each other via wired networks. Therefore, those algorithms usually suffer from heavy networking costs coming from frequent data transits over wireless networks, if they are used in the wireless computing environment. In this paper, to reduce usage of wireless communications, our checkpointing algorithm allows the distributed mobile application to tune the level of its checkpointing strictness. The strictness is defined by the maximum rollback distance (MRD) that says how many recent local checkpoints can be rolled back in the worst case. Since our algorithm have more flexibility in checkpointing schedule due to the use of MRD, it is possible to reduce the number of enforced local checkpointing. In particular, the amount of data transited on wirelesses networks becomes much smaller than in earlier methods; thus, our algorithm can provide less communication cost and shortened blocking time.

Keywords: *Mobile networks, distributed application, rollback, recovery, distributed checkpointing.*

1. Introduction

During the past decades, there have been dramatic advances in mobile networks and mobile devices. In particular, the fast spreading usage of smart phones is likely to yield demands for sophisticated distributed applications across multiple mobile devices [1, 2]. During the run-time of such a distributed application, its cooperating application processes (APs) work in parallel and data are usually transited between APs to share application contexts. In this situation, failure on a single AP or hardware device could cause a serious problem in the whole distributed application and thus it may roll back the application's processing state to the initial one in the worst case. To prevent a whole cancelation of the processing result, checkpoint records are created to log intermediate execution results. The recovery procedure after abrupt failure builds a consistent state of an application from the checkpoint data, and resumes the interrupted application from that state. This can reduce undesirable loss of application process

To make the distributed application robust and recoverable against failure, many works are done for the computing environments where the distributed application seems to be executed in the wired fixed networks [4, 6, 7, 10, 11, 13]. When checkpointing algorithms of those earlier works are applied to distributed applications running on wireless networked, they suffer from high cost for sending checkpoint data via warless connections. Since data transit over wireless networks is more costly and unstable, compared with that over wired networks, many researches focus on reduction of wireless data transit in the case of the checkpointing scheme for wireless computing environment [5, 8, 9,10, 12].

In the paper, we also propose a distributed checkpointing scheme suitable for distributed applications running on the mobile computing environment. We here introduce two key ideas of the maximum rollback distance (MRD) and the logging agent running on the MSS (Mobile Support Station). The logging agent is a software agent running on MSS, which is responsible for making local checkpoints at the request of its associated AP's requests and maintaining at least one consistent global checkpoint. To save the cost for maintaining such a global checkpoint, the agent can do some logging activities without any requests from associated AP. For this, the logging agent securitizes messages arriving on its MSS and communicates with other logging agents for synchronization of checkpointing.

If the rule of checkpointing synchronization is too strict, enforced local checkpoints are frequently created. Sine enforced local checkpoint request costly data transit in wireless network lines; it is needed to make the synchronization rule more flexible. In this notion, we introduce the MDR for each distributed application. On the other hand, the MDR is a run-time parameter for a distributed application, saying how many local checkpoint of a given AP can be rolled back in the worst case. With a MDR properly set to a value, a significant flexibility is available at the time local checkpoints are synchronized in order to create a new global checkpoint. Due to the tunable level of checkpointing synchronization using MDR, the logging agents participating in a distributed application can reduce the number of enforced local checkpointing and costly message transit over wireless networks.

The rest of this paper is organized as follows. In Section 2, we describe some backgrounds regarding the meaning of global consistency of distributed checkpoints, the assumed mobile network, and the previous works. Then, we propose a new efficient distributed checkpointing scheme in Section 3, and discuss the performance characteristics of our scheme in Section 4. Lastly, we conclude this paper in Section 5.

2. Backgrounds

2.1 Global Consistent State

The GCS (Global Consistent State) of distributed applications was formally defined by Lamport [17]. According to that definition, processing of a distributed application can be modeled by three types of events such as the message sending event, the message receiving event, and the computation event. Each AP participating in a distributed event can do the computation event to proceed with its processing state and communication with other participant APs through message sending/receiving events.

In this event model, a set of events meeting the GCS can be captured using the relation "happen-before" drawn on events. In [17], the "happen-before" relation (HBR) is as follows.

[Definition of HBR] If it is the case that *e1* "*happen-before*" *e*2, then either of the following conditions should be true.

i) Both *e1* and *e2* occur in the same AP and *e1* precedes *e2* in timing sequence.

ii) There are a message m and two APs of *p1* and *p2* such that *p1* sends (event *e1*) message *m* to *p2* and *p2* receives (event *e2*) it.

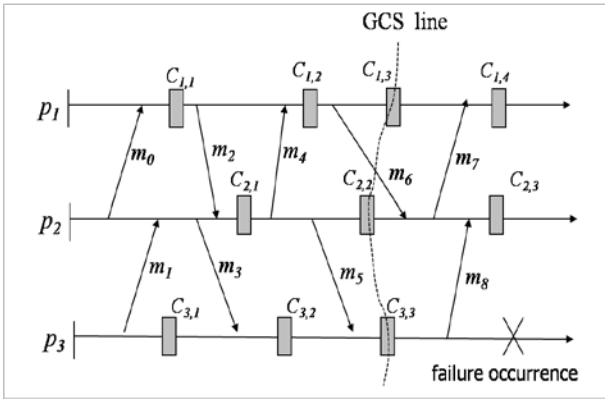


Fig. 1. An example of GCS: a failure arises at *p3*.

Owing to the transitive property of the HBRs, we can give a partial order to the events of a distribute application, even though there is no common clock shared by its participant AP's.

The GCS of a distributed application is defined based on the HBR above. Let us take a snapshot of execution state of a distributed application at a particular time, and let *S* be that snapshot, which is a set of the events having arisen in the application. Let *G* be a subset of *S*. In this case, *G* is said to be in a GCS if the following condition is satisfied; for every event *e'* in *G*, if there is *e* in *S* such that *e* "happen-before" *e'*, then *e* should be also an event in *G*. In other words, for every event of *G*, its causal events should be found in *G*. Since all the causal events are contained in G, it may be possible to obtain the same execution results of G, if we redo the events of G from it begging time. Based on this idea, we can recover any intermediate execution state of any failed application if its any GCS execution snapshot is available.

To have execution snapshots, checkpointing schemes are commonly used for saving local execution state of individual AP's. Fig.1 shows an example where distributed checkpoint is performed by three AP's, *p1*, *p2*, and *p3*. In the figure, the blacked rectangle of $C_{i,k}$ represents the *k*-th local checkpoint made by AP p_i. The local checkpoint of $C_{i,k}$ is made to save the computational computation state of p_i and the message sent to other AP's after the creation time of $C_{i,k-1}$.

Suppose that an application failure arise at *p3*. as in Fig. 1. At this moment, the set of local checkpoints preserving the GCS are that inside the GCS line of the figure. That is, the latest CGS state is composed of $C_{1,3}$, $C_{2,2}$, and $C_{3,3}$. As the message sending event of m8 is not saved in any local checkpoint, its message receiving event cannot be include a GCS. Therefore, $C_{2,2}$ is rolled back, and $C_{1,3}$ is also rolled back because $C_{1,3}$ contains the message-receiving event of m_7 saved in $C_{2,2}$. As a result, the local

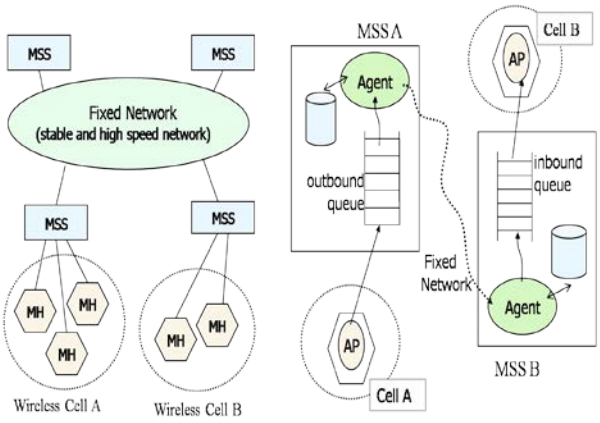
checkpoints on the GCS line will be used to recover the failed application.

In the example of Fig. 1, the latest local checkpoints of a GCS are the same as $C_{1,3}$, $C_{2,2}$, and $C_{3,3}$ is used as a latest consistent global checkpoint. This choice of such a consistent global checkpoint is done by a recover algorithm initiated in the presence of failure. During the recovery phase, such a latest consistent global checkpoint is found and then the disrupted distributed application is restarted. In the case of Fig. 2, the associated AP's will restore their computational state using the data saved in the local checkpoints of $C_{1,3}$, $C_{2,2}$, and $C_{3,3}$, respectively, and *p1* will send the lost message m_6 to *p2* again. Since checkpoint records bookkeep the serial numbers of messages transferred among AP's, this message resending is possible.

2.2 Assumed Mobile Network

In general, the mobile network is comprised of mobile hosts (MH's), mobile support stations (MSS's), and the fixed networks interconnecting the MSSs [1, 2, 8]. The network architecture is shown in Fig. 2(a), where there are two wireless cells and MH's can make wireless network connections within its wireless cell. Since the MH can hop among wireless cells, the MSS's have to update the list of MH's under control for seamless hand-offs. Each AP can be identified by unique process id within its hosting MH. Of cause, the MH is also uniquely identified in the global network environment.

Consider a situation where a MH x in MSS A sends a network message m to an MH y in MSS B. The message m from x is queued into an outbound queue of MSS A and then it is delivered to the counterpart MSS B via the fixed network. Consecutively, message m is entered into the inbound queue of MSS B for the delivery towards y.
On the top of the traditional architecture of Fig. 2(a), we assume that an agent program executes on each MSS for doing checkpoint-related activities. That is, it is assumed that the agent program makes accesses the two message queues of the MSS, in which outbound or inbound network messages are temporarily stored waiting for their delivery to target AP. Fig. 2(b) depicts the assumed architecture with logging agents. In the example of Fig. 2(b), an MH in cell A sends a network message m to other MH staying in cell B. In this case, the logging agent in MSS A dequeues message m and then appends some checkpoint-related data to m before it sends m to the logging agent of MSS B. Correspondingly, the logging agent of MSS B deletes the



(a) Traditional network architecture. (b) Assumed network architecture.

Fig. 2. Architecture of the assumed mobile network.

appended data from m before it inputs m into the inbound queue towards the destination MH. During this message transit time, the logging agent's can make checkpoint records in the disk storage installed in the MSS's. Using the logging agent, we can reduce the checkpoint cost and improve the flexibility of the consistent global checkpoints. The more details about the logging agent are described in Section 3.

2.3 Earlier Works

Checkpointing schemes for distributed applications can be roughly categorized into the synchronized schemes and the asynchronized schemes. In the synchronized schemes, when a AP requests a checkpoint, actions for making a consistent global checkpoint are performed such that the newly created global checkpoint includes the current execution state of the checkpoint-requesting AP. From this, the checkpoint-requesting AP can make its crucial results of execution robust to any failure. For such checkpointing, the checkpoint requester AP is blocked until all the causal events of the checkpoint-requested events are saved in the local checkpoint records of the participant AP's. Owing to such creation of a global checkpoint, most of execution results can be restored in the present of failure.

However, because creation of a global checkpoint needs a number of message deliveries and requires some enforced checkpointing of other participant AP's, this scheme suffer from a high network cost and long delay time for checkpointing. Especially, such shortcomings become more serious in the case where the AP's are ruing in mobile network environment [1, 2, 8, 12].

Meanwhile, the asynchronized scheme does not enforce the creation of a consistent global checkpoint at every checkpoint request time. Instead, during the recovery phase a latest consistent global checkpoint is found from the casual dependency of APs' events saved in disturbed local checkpoints. By examining the casual dependency among the previous local checkpoints, the recovery algorithm picks a most recent consistent global checkpoint for recovery. Since each AP can make its local checkpoint in asynchronized manner, this scheme is apt to have a problem of many cascaded rollbacks of local checkpoints, so-called domino effect [1, 8, 17]. In a worst case, the whole execution results of any distributed application can be cancelled because of the domino-effect. In addition, the asynchronized schemes have more restarting overheads, compared to the synchronized schemes. This is because the asynchronized scheme has to collect the whole information from scatted local checkpoints in order to find a consistent global checkpoint. From these reasons, the synchronized scheme is preferred in earlier time.

However, when it comes to the mobile network environment, the synchronized scheme is more feasible because of less network connectivity and more consideration of instability of mobile devices of that application environment. Among the asynchronized schemes, in particular, the message-induced checkpointing scheme [8, 9] is regarded to be a good alternative solution in the mobile network environment. This is because the message-induced scheme can eliminate the possibility of domino effect sin a very simple manner. By forcing AP's to make local checkpoints depending on the message-receiving events, this scheme can set some boundaries on rollbacked local checkpoints

3. Proposed Method

3.1 Motivations

Although the message-induced scheme is useful to avoid the domino effect, it has a severe problem in that the time of checkpoint creations is determined without active involvement of participant AP's. This problem can be easily understood by viewing the used mechanism of the message-induced scheme. The scheme uses two different execution states of the AP, that is, SEND and RECEIVE states. The SEND and RECIVED states are set while the message-sending and message-receiving events are successively arising, respectively. A new checkpoint is compulsorily created at the time when a network message arrives at a particular AP with the execution state of SEND. Since the time of checkpoint creations in an AP is inactively determined depending on the arrivals of message-receiving events, the created checkpoints would not reflect application's semantics. Otherwise, if we want application's semantics-aware checkpointing, that checkpointing time of checkpoint can be chosen by considering the critical points of processed application. In other words, checkpoints have to be made when some critical executions or expensive processes are done. Such semantics-aware checkpointing is possible only when the AP can actively request checkpointing. Note that the user can also issue checkpoint requests via its AP.

The lack of semantics-awareness of the message-induced scheme may have poor performance. When this scheme is used for distributed mobile applications requiring a lot of message transit, a large number of non-meaningful checkpoints can be made. This can result in frequent checkpointing and creations of obsolete checkpoints. In addition, the message-induced scheme has no mechanism to actively create consistent global checkpoints. Therefore, in the case that an AP wants to make its critical execution results persistent, there is no way for that. Whether or not the execution results are saved into a consistent global checkpoint relies on the existence of an appropriate pair of message-receiving and message-sending events.

To solve such problems of lack of semantics-awareness in checkpointing time and its defective mechanism for global checkpointing, we propose a new checkpoint scheme based on a combination of the logging agent and the R-distance

3.1 Data Format

First, we describe the data format of the network message, which is represented by **M** below. In the followings, the message sender AP is denoted by P_i and the total number of AP's joining the distributed application by N, respectively. The fields of **M** are seven in all. Among them, the last three fields are not used by the AP. These fields exist for containing control data of checkpointing purpose and they are visible only to the logging agent. Meanwhile, the first four are for containing application data needed by the AP.

- **M**.*type*: Message type
- **M**.*sender*: *Id* of P_i
- **M**.*recever*: *Id* of the counterpart AP
- **M**.*data*: Application data sent to the counterpart AP
- **M**.*ap*[*1,2,…,N*]: Ids of AP's joining this application.
- **M**.*serial*[*1,2,…,N*]: Serial numbers of the local checkpoints already made by **M**.*ap*[*1, 2, …N*].
- **M**.*dep_vec*[*1,2,…,N*]: Checkpoint dependency vector

To make the checkpoint-related fields invisible to the entire participant AP's, the logging agent appends these fields at the tail of any message *m* received from its AP before it sends message *m* to other logging agent. Then, the receiver logging agent will delete these fields from *m* before *m* is sent to the destination AP. The dependency vector used for finding a consistent global checkpoint is the same as that proposed in the earlier literatures [7, 8, 9, 11]. For space limitation, the details of use of the dependency vector are referred to the literatures.

In turn, we describe the data format of the checkpoint record managed by the logging agent. The fields below are ones existing in the checkpoint record. When an AP initiates or joins a distributed application, a logging agent of the AP creates a new checkpoint record for saving transferred network messages and checkpoint-related data until the next checkpointing time. At the creation time, the checkpoint record is manipulated in an area of main memory, and then it is written into a stable storage at the next checkpointing time. In the followings, the owner AP of the checkpoint record is represented by P_i, and the total number of AP's joining the distributed application by *N*.

- **REC**.*id*: Id of P_i.
- **REC**.*serial*: Current checkpoint serial number
- **REC**.*r_distance*: R-distance of this application.
- **REC**.*ap*[*1,..,N*]: Ids of participant AP's
- **REC**.*serial*[*1,..,N*]: Serial numbers of the local checkpoints already made by **REC**.*ap*[*1, 2, …N*].
- **REC**.*dep_vec*[*1,..,N*]: checkpoint dependency vector
- **REC**.*message*[]: Messages sent by P_i after the last checkpointing time
- **REC**.*prev_rec*: Disk address to the previous checkpoint record

Fig. 3. logging agent algorithm for handling a message-receiving event.

As known from the above, at the first four fields the checkpoint record saves the *id* of the owner AP, the serial number of the current checkpoint record, the given R-distance, and *id*s of the participant AP's. And, the next two fields are used for bookkeeping the information about created local checkpoint serials and dependency vector.

To log all the network messages sent by P_i until the next checkpointing time, we use the field of *message[]*. Since all the network messages sent to other AP's are logged in that field, messages resending can be done during the recovery phase. Since more than one checkpoint record are created o be created for the same application while the application executes, they are chained for the fast access during the recovery time. The last field is used for that purpose, that is, it saves the disk address to the very previous checkpoint record stored in the disk.

3.2 Tunable Checkpointing

The semantics-aware checkpointing requires that local checkpoints be made according to the determination to importance of the current execution state. Here, the expensive processing is some actions whose loss causes many additional network communications or computational overheads. Such expensive processing is according to application semantics, and thus only the involved AP is responsible for its determination. For that reason, the capability of creating a global checkpoint by the AP is needed, as supported in the synchronized checkpoint scheme. However, such capability inevitably results in a high network cost and long blocking-time when the protocol of the previous synchronized checkpointing schemes is applied to the mobile computing environment.

USED DATA: *R* /* current checkpoint record of P_i */

When a message *m* arrives at C_i from P_i

1. **begin**
2. **if** (*m* is for requesting a checkpoint creation) **then**
3. Save the content of *R* into disk space to make a local checkpoint with the serial number of *R*.serial.
4. *R'* ← GetGCSRec(*R*). /* get a latest checkpoint record of a GCS */
5. **if** (*R'*= **nil**)
6. Call the routine *GreateGCS*(*R*).
7. **endif**
8. Create a new checkpoint record with the serial number of *R*.serial + 1.
9. Send a messages notifying the creation of a new local checkpoint of P_i.
10. Send a response message of checkpointing to P_i .
11. **else** /* *m* contains application data sent to other AP */
12. Append checkpoint-related fields to *m* and send it to the counterpart logging agent.
13. **endif**
14. **end.**

When a message *m* arrives at C_i from other logging agent

15. **begin**
16. **if** (*m* is an application data message toward P_i)
17. Call the routine *UpdateChptRec*(*m* , *R*).
18. Remove some checkpoint-related fields form *m* and sent it to P_i .
19. **else if** (*m* is for notifying creation of a new remote checkpoint)
20. *UpdateChptRec*(*m*, *R*).
21. **else** /* *m* is for requesting P_i 's checkpointing */
22. Make a message for requesting an enforced checkpointing and set it to P_i .
23. **endif**
24. **end**.

Fig. 3. logging agent algorithm for handling a message-receiving event.

To solve such a problem, we introduce the notion of the recovery distance (R-distance) for the distributed application. The R-distance indicates the worst-case number of rollbacked checkpoints in the presence of failure. If its value is *d*, then the latest *d* -1 local checkpoints can be rolled back at the worst-case. For example, if its value is equal to three, then the latest two checkpoints can be rolled back with respect to each participant AP. In the same way, if its value is one, our scheme will work identically with an earlier synchronized checkpoint scheme, where every checkpoint request results in creation of a new consistent global checkpoint. If the current application is in a very mission critical state, then the application initiator AP can set the R-distance to a small one. Additionally, if an AP really wants to make a global checkpoint for a distributed application with R-distance *d*, it can do that by issuing *d* local checkpoint requests successively.

The R-distance value is determined and assigned to every distributed application at its beginning point, and the global checkpoints are made in flexible manner, while preserving the given R-distance. In our scheme, the enforced global checkpoint is issued by only the logging agent and the necessity of such enforced checkpointing is also decided by the logging agent. The AP just issues a request for creating its local checkpoint by reflecting application semantics.

The algorithm of Fig. 3 shows the way a logging agent works at the time when a network message *m* arrives at the logging agent. In the algorithm, the logging agent receiving message *m* is denoted by C_i, and the AP checkpointed by C_i is denoted by P_i. The message *m* can be one from P_i or any other logging agent. Since all the messages sent to an AP are relayed by logging agent's, every message from AP's other than P_i arrives at C_i via the logging agent's.

The steps of lines 1–14, are executed if C_i receives a message from P_i. In this case, C_i first checks if message *m* is for requesting a creation of P_i's local checkpoint. If that is true, the steps of lines 3-9 are performed to make a new local checkpoint, preserving the R-distance of the distributed application. For this, C_i calls the routine *GetGCSRec*() to get the lasts checkpoint record of a GCS. Then, the record's serial number is compared with the that of the newly created local checkpoint. If preservation of the R-distance constraint is not possible, then the routine *CreateGCS*() is executed to make a new consistent global checkpoint as in line 6. Otherwise, if the R-distance is preserved, then C_i just saves the current checkpoint record and send a response message back to P_i for notifying successful checkpointing. On the other hand, if *m* is a pure application data message, then the message is delivered to the counterpart logging agent managing the message receiver AP. At that time, some fields used for checkpointing are appended to the original *m*.

First, we describe the data format of the network message, which is represented by **M** below. In the followings, the message sender AP is denoted by P_i and the total number of AP's joining the distributed application by *N*, respectively. The fields of **M** are seven in all. Among them, the last three fields are not used by the AP. These fields exist for containing control data of checkpointing purpose and they are visible only to the logging agent. Meanwhile, the first four are for containing application data needed by the AP.

The rest steps in lines 15-24 of Fig. 3 are ones to be performed when C_i receives *m* from other logging agent, say C_j. If *m* is for sending application data to P_i, then it is sent to P_i, after some piggybacking fields are deleted from *m*. Of course, to save the checkpoint-related data the routine *UpdateChptRec*() is called in line 17. If message *m* is not for sending pure application data, it is either for notifying a new checkpoint creation in the side of C_j or for forcing P_i to create a new local checkpoint. In the former case, C_i just updates the current checkpoint record for reflecting the advance of the remote checkpoint serial number and other changes of the distributed application. Since M has no application data in itself, further message delivery is not needed. In the latter case, a new message forcing P_i to make its local checkpoint is sent to P_i as in line 22. In the response of that message, P_i will send a message for checkpoint creations, and then line 3 is executed later.

The main advantages of our checkpointing scheme in Fig. 3 are two-fold. First, based on the concept of R-distance, the average cost for creating a consistent global checkpoint can be reduced, because creation of the global checkpoint can be delayed within the R-distance. Additionally, if no global checkpoint is found within the R-distance, then our scheme estimate the costs of global checkpoints within R-distance in routine *CreateGCS*(). Those features differentiate our checkpointing protocol from others used for global checkpointing in the earlier schemes, which only have to create a global checkpoint containing the latest local checkpoint without any consideration of its creation cost. With our flexibility and cost estimation in global checkpoint time, we can reduce the average cost for making a global checkpoint.

Second, the use of the logging agent can reduce the amount of checkpoint-related data transferred between MSS's and MH's. Since those additional data for checkpoint is always needed for tracking application's execution state,

```
Algorithm: Routine GetGCSRec(R)
INPUT:  Pi's checkpoint record R
OUTPUT: checkpoint record having a GCS

1.  A[1] ← R.
2.  for i =2 to R.r_distance
3.    A[i] ← ReadChptRec(A[i-1].previous). /* reading of the
      previous checkpoint records */
4.  endfor
5.  p_num ← number of AP's saved in R.ap[].
6.  for i = 1 to R.r_distance
7.     cgs_exist ← yes.
8.     for j = 1 to p_num
9.        if ( A[i].chpt_dep_vec[j] > R.serial[j] )
10.          cgs_exist = no.
11.    endfor
12.    if( cgs_exist = yes ) return A[j].
13. endfor
14. return nil.  /* no GCS checkpoint record */
```

Fig. 4. Algorithm for routine *GetGCSRec*().

overheads for sending them are unavoidable. If the communication overheads become too large on wireless networks, that could be a severe bottleneck in the execution of the distributed applications. Against such a problem, we adopt the logging agent so that most of the additional checkpoint-related data are visible only in the network messages transferred within logging agent's wired fixed networks. Since the communication cost in the fixed networks of logging agent's is very lower than in wireless communication, we can reduce the overall network cost due to the use of logging agents.

3.2 Detailed Algorithms

In Fig. 3, we outlined the proposed algorithm of the logging agent. In that algorithm, we omitted details of the routines called by the logging agent in Fig. 3. Here, we present the detailed algorithms of the routines. Besides those routines of Fig. 3, other routines used by the AP are also needed for our checkpointing scheme. For instance, we need the routines for message sending/receiving, requesting a local checkpoint, and processing a checkpoint enforcement message. As the algorithms for those AP routines are not distinctive from ones previously proposed in [8, 9] and they can be conjectured from the algorithm of the logging agent, we do not present them in this paper.

Fig. 4 depicts the algorithm of routine *GetGCSRec*() used to get a latest checkpoint record of P_i being in a GCS. Here, P_i is the AP whose checkpoint record is *R* of this routine. In lines 1-4, the routine reads the previous checkpoint records into the memory areas of *A*[2], .., *A*[*R*.r_distance] and the current checkpoint record into *A*[1], respectively. For this, the backward pointers chaining the disk-resident checkpoint records are used for fast accesses. Using the dependency vector and the serial local checkpoint numbers saved in *A*[1, …, *R*.r_distance], this routine finds a latest GCS.

The algorithm of routine *GetGCSRec*() is based on the concept of the local checkpoints dependency among different AP's. This is represented by the dependency vector saved in the checkpoint record field of *dep_vec[]* of Fig. 3. The use of dependency vector is common in the global checkpoint schemes [5, 6, 9, 10, 11]. The proof on the usefulness of the dependency vector is also referred to these researches.

We also use the dependency vector for deterring a collection of local checkpoints with a GCS, that is, a consistent global checkpoint. This routine compares the dependency vector saved in R the current serial number of the latest checkpoint records of other AP's. The latest checkpoint serials are found in the data structure of *serial[]* in *R*. With the comparison, this routine can finds the latest local checkpoint whose dependency are checked that is not dependent on the events that have not been check-pointed by counterpart AP's.

In lines 6-13, the logging agent decides whether or not the current creation of a local checkpoint supports the R-distance preservation. If its preservation is not possible due to the current checkpoint request, the routine *CreateGCS*() is called as in line 6 of Fig. 3.

Fig. 5 depicts the algorithm of routine *CreateGCS*(). This routine also reads the previous checkpoint records into *A*[] for fast manipulation. Then, in lines x-x the routine computes the global checkpoint costs with respect to the local checkpoints represented by *A*[]. Here, the costs are assessed by the number of remote local checkpoints to be created for yielding a GCS, plus the distance of the chosen local checkpoint from the current checkpoint time. That is performed in lines 8-15 of Fig. 5. Based on the estimation, the forced GCS line is determined by favorably choosing a local checkpoint with the smallest costs as in line 16.

To make the chosen local checkpoint, denote by *A*[*s*] in the algorithm of Fig. 5, be a global consistent one, messages for requesting enforcement of checkpointing in other AP's are sent to the involved logging agent's. Then, the logging agent's will enforce its AP to create the local checkpoint. When all the response messages are gathered, this routine returns. ,

The routine *UpdateChptRec*() of Fig. 6 is for modifying the checkpoint record in accordance with message arrivals. In line 2, the routine check if m is an outbound message

Algorithm: Routine *CreateGCS(R)*
INPUT: *Pi's* checkpoint record *R*

1. *A*[1] ← *R*.
2. **for** *i* =2 **to** *R*.r_distance
3. *A*[*i*] ←*ReadChptRec*(*A*[*i-1*].previous). /* reading of the previous checkpoint records */
4. **endfor**
5. *p_num* ← number of AP's saved in *R*.ap[].
6. *needed_local*[1,..,*R*.distance] ← 0. /* number of local checkpoints created for making a GCS */
7. *costs*[1,..,*R*.distance] ← 0. /* initialization */
8. **for** *i* = 1 **to** *R*.r_distance
9. **for** *j* = **to** *p_num*
10. **if** (**A**[*i*].dep_vec[j] > *R*.serial[j])
11. *needed_local*[[*j*]++.
12. **endif**
13. **endfor**
14. *costs*[*i*] = *i* + *needed_local*[*i*].
15. **endfor**
16. Find the least element among *costs*[1], *costs*[2],.., *costs*[*R*.distance] and let *s* be the index of that element.
17. **forall** AP p such that *A*[*s*].dep_vec[p] > *R*.serial[p]
18. Send a checkpoint requesting message to the logging agent managing the checkpoint record of *p*.
19. **endfor**
20. Blocked until all the response messaged are received form the logging agent's above.

Fig. 5. Algorithm for routine *CreateGCS*().

sent by Pi. If that is true, the routine saves the content of m in the checkpoint record being located in memory. Otherwise, if m is an inbound message, that is, m is a message coming from other logging agent, then the data of checkpoint's dependency and the serial numbers of the local checkpoints of other AP's are updated to reflect the changes of in the distributed application state.

4. Performance Analysis

To analyze the performance of the proposed scheme, we consider two key metrics, that is, less overhead paid for making checkpoints during normal execution time and low possibility of rollbacked executions in the face of application failure. Because there is a trade-off between these two metrics and they are largely affected by diverse factors such as application or network failure rates and frequency of checkpoint creations, it is very hard to devise an exact performance metrics. For these limitations, we give only a rough analysis on our checkpointing scheme here. ,

We first look on the overhead paid to generate checkpoint records during normal time. The main components of such overhead cost seem to be AP's blocking time for checkpointing and communication costs for sending

Algorithm: Routine *UpdateChptRec(m, R)*
INPUT: received message *m*, *Pi's* checkpoint record *R*

1. **if**(*m* is an outbound message) /* heading for other AP */
2. Save *m* into *R* and advance the number of sent messages by one.
3. **else** /* *M* is a message arriving at *Pi* */
4. **foreach** p in *R*.ap[] /* participant application processes */
5. *R*.serial[p] ← max(*m*.serial[p], *R*.serial[p]).
6. *R*.dep_vec[p] ← max(*m*.dep_vec[p], *R*.dep_vec[p]).
7. **endfor**
8. **endif**

Fig. 6. Algorithm for routine *UpdateChptRec*().

additional data used for tracking execution states of ongoing application. The blocking time in our algorithm is very short on average, compared with the traditional algorithms. To see that, recall the steps of lines 5-7 in Fig . 3. In those steps, blocking time arises only when a globally consistent local checkpoint is not found within the R-distance. In many cases, such a situation is not the case. Even though there is a need for creating a new global checkpoint, our protocol will choose a local checkpoint whose checkpointing overhead is most cheap. That is done by the logging agent by using the routine *CreateGCS*() of Fig. 5. Using this routine, the logging agent can choose among previous local checkpoints any one that demands a least number of local checkpoints.

The network cost for checkpointing depends on the amount of additional checkpoint-related data that piggybacks on messages delivering application data. In particular, such data should be less on the warless communications. To reduce the additional data on wireless networks, the logging agent manages checkpoint-related information using its checkpoint record in memory and refers to that for generating messages being transferred among logging agent's. From this, additional network overheads for checkpointing are small in our scheme, because most of additional data are not visible to AP's,

In the aspect of less cancellation of execution results in the case of application failure, our algorithm has a good property. Due to R-distance, the number of rollbacks in a particular AP is always less than a value set to R-distance. To all AP's participating in the application, the worst case number of rollbacks is less than $N \times (d\text{-}1)$ while N AP's are running with R-distance of d. That is, there is a tight upper bound on the number of cancelled local checkpoints. However, such a large cancellation is not realistic, since the times of checkpoint creations are different among the AP's joining a distributed transaction. In probabilistic, the

average number of rollbacked local checkpoints remains below a half of the upper bound number, that is, $N \times (d\text{-}1)/2$. Therefore, by setting R-distance appropriately, we can give a limit on the losses of execution results. From these properties, we can say that our algorithm can have a less cancellation of application by setting R-distance in a low range. Consequently, the proposed method has advantages during the normal execution time and recovery phase.

5. Conclusions

The problem of making a consistent distributed checkpoint in a mobile network environment is challenging to solve. This is because the distributed application needs application processes' communication via warless networks and such wireless communications easily make the cost for checkpoint higher. To have less communication cost, the previous checkpoint schemes for mobile distributed applications take an approach to making local checkpoints in a very inflexible manner. Such inflexible in the checkpointing causes the lack of semantics-awareness in the time of checkpointing. In addition, some asynchronized checkpointing scheme cannot provide any efficient mechanism for global checkpointing by the application process. These shortcomings can make obsolete checkpoints and frequent losses of expensive execution results. To solve those problems, we proposed a new checkpoint scheme based on the checkpoint agent and the concept of the recovery distance. From the combination of them, the proposed scheme provides the capability of semantics-aware checkpointing by paying only a cheap cost. We believe that the proposed checkpointing scheme can be applied to recover the mobile distribute application from diverse failures.

References

[1] T. Imielinski and B. R. Badrinath, Mobile Wireless Computing: Challenges in Data Management,Communications of the ACM, pp.19-28, Vol.37, No.10, October 1994.

[2] Yi-Bing Lin, Failure Restoration of Mobility Databases for Personal Communication Networks, Wireless Networks, Vol.1, No.3, 1995.

[3] Sashidhar Gadiraju and Vijay Kumar, Recovery in the Mobile Wireless Environment Using Mobile Agents,IEEE Trans. on Mobile Computing, Vol.3, No.2, April 2004.

[4] Ricardo Baratto, Shaya Potter, Gong Su, and Jason Nieh, MobiDesk: Mobile Virtual Desktop Computing, In Proc of the 10th International Conference on Mobile Computing and Networking, pp.1-15, 2004.

[5] Dhiraj K. Pradhan, P. Krishna, and Nitin H. Vaidya,Recovery in Mobile Wireless Environment: Design andTrade-off Analysis, In Proc. of the 26th International Symposium on Fault-Tolerant Computing, pp.16-25,1996.

[6] Arup Acharya and B. R. Badrinath, Checkpointing Distributed Applications on Mobile Computers, In Proc. of the 3rd International Conference on Parallel and Distributed Information Systems, pp.73-80, 1994.

[7] Y. M. Wang, Consistent Gobal Chckpoints That Contain a Given Set of Local Checkpoints, IEEE Trans. on Computers, Vol.46, No.4, pp.456-468, 1997.

[8] Tongchit Tantikul and D. Manivannan, Communication-Induced Checkpointing and Asynchronous Recovery Protocol for Mobile Computing Systems, In Proc. of the 6th International Conference on Parallel and Distributed Computing Applications and Technologies, pp.70-74, 2005.

[9] Taesoon Park and Heon Y. Yeom, An Asynchronous Recovery Scheme based on Optimistic Message Logging for Mobile Computing Systems, In Proc. of the 20th International Conference on Distributed Computing Systems, pp.436-443. 2000.

[10] R. E. Strong and S. Yemini, Optimitic Recovery in Distributed Systems, ACM Trans. on Computer Systems, Vol.3, No.3, August 1985.

[11] D. Manivannan and Mukesh Singhal, Quasi-Synchronous Checkpointing: Models, Characterization, and Classification, IEEE Trans. on Parallel and Distributed Systems, Vol.10, No.7, July 1999.

[12] Cheng-Min Lin and Chyi-Ren Dow, Efficient Checkpoint-based Failure Recovery Techniques in Mobile Computing Systems, Journal of Information Science and Engineering, pp.549-573, Vol 17, No.4, 2001.

[13] Lorenzo Alvisi, E. N. Elnozahy, Sriram Rao, Syed Amir Husain and Asanka De Mel, An Analysis of Communication Induced Checkpointing, In Proc. of the Symposium on Fault-Tolerant Computing Symp., pp.242-249, 1999.

[14] Franco Zambonelli, On the Effectiveness of Distributed Checkpoint Algorithms for Domino-Free Recovery, In Proc. of High Performance Distributed Computing,pp.124-131, 1998.

[15] Mootaz Elnozahy, et. al., A Survey of Rollback-Recovery Protocols in Message-Passing Systems, Technical Report: CMU-CS-99-148, June 1999.

[16] Yi-Min Wang and W. Kent Fuchs, Lazy Checkpointing Coordination for Bounding Rollback Propagation, In Proc. of the International Symposium on Reliable Distributed Systems, pp.78-85, 1993.

[17] Lapmport, Time, clocks, and the Ordering of Events in a Distributed System, Communication of ACM, Von. 21, No.7, pp.558-565, 1978.

Sungchae Lim received the B.S. degree in Computer Engineering form Seoul National University at 1992, and achieved the M.S. and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), at 1994 and 2003, respectively. He also worked for the Korea Wisenut Cooperation from 2000 to 2005, and he is currently an Associate Professor in the Department of Computer Science at Dongduk Women's University. His research interest includes the high-performance indexing, mobile computing, and semantic Web.