Hierarchical Non-blocking Coordinated Checkpointing Algorithms for Mobile Distributed Computing

Surender Kumar ssjangra20@rediffmail.com
Deptt. of IT,
H.C.T.M
Kaithal (HRY), 136027, INDIA

Parveen Kumar pk223475@yahoo.com
Deptt. of CSA,
M.I.E.T
Meerut (U.P), INDIA

R.K. Chauhan rkc.dcsa@gmail.com
Deptt. of CSA,
K.U.K
Kurukshetra (HRY), INDIA

Abstract

Mobile system typically uses wireless communication which is based on electromagnetic waves and utilizes a shared broadcast medium. This has made possible creating a mobile distributed computing environment and has brought us several new challenges in distributed protocol design. So many issues such as range of transmission, limited power supply due to battery capacity and mobility of processes. These new issue makes traditional recovery algorithm unsuitable. In this paper, we propose hierarchical non blocking coordinated checkpointing algorithms suitable for mobile distributed computing. The algorithm is non-blocking, requires minimum message logging, has minimum stable storage requirement and produce a consistent set of checkpoints. This algorithm requires minimum number of processes to take checkpoint.

Keywords: Co-ordinated Checkpointing, fault tolerant, Non-blocking approach, Mobile Computing System.

1. INTRODUCTION

The market of mobile handheld devices and mobile application is growing rapidly. Mobile terminal are become more capable of running rather complex application due to the rapid process of hardware and telecommunication technology. Property, such as portability and ability to connect to network in different places, made mobile computing possible. Mobile computing is the performance of computing tasks whiles the user in on the move, or visiting place other than their usual environment. In the case of mobile computing a user who is away from his “home” environment can still get access to different resources that are too computing or data intensive to reside on the mobile terminal [4].Mobile distributed systems are based on wireless networks that are known to suffer from low bandwidth, low reliability, and unexpected disconnection [3].

Checkpointing / rollback recovery strategy has been an attractive approach for providing fault tolerant to distributed applications [1] [16]. Checkpoints are periodically saved on stable storage and recovery from a processor failure is done by restoring the system to the last saved state. So the system can avoid the total loss of the computation in case of the failure. In a distributed system, since the processes in the system do not share memory, a global state of the system is defined as a set of local states, one from each process. An orphan message is a message whose receive event is recorded, but its sent event is lost. A global state is said to the "consistent" if it contains no orphan message and all the in-transit messages are logged. To recover from a failure, the system restarts its execution from a previous consistent global state saved on the stable storage during fault-free execution. This saves all the computation done up to the last checkpoint state and only the computation done thereafter needs to be redone [7], [12], [13]. Synchronous and asynchronous are two fundamental approaches for checkpointing and recovery [2].

In uncoordinated or independent checkpointing, processes do not coordinate their checkpointing activity and each process records its local checkpoint independently [8], [14], [15]. After a failure, a consistent global checkpoint is established by tracking the dependencies. It may require cascaded rollbacks that may lead to the initial state due to domino-effect [11], [12], [13].

In coordinated of synchronous checkpointing, processes take checkpoints in such a manner that the resulting global state is consistent. Mostly it follows two-phase commit structure [9], [10], [11]. In the first phase, processes take tentative checkpoints and in the second phase, these are made permanent. The main advantage is that only one permanent checkpoint and at most one tentative checkpoint is required to be stored. In case of a fault, processes rollback to last checkpointed state. A permanent checkpoint can not be undone.

Coordinated checkpointing algorithms can be blocking and non blocking [3]. A primitive is blocking if control returns to the invoking process after the processing for the primitive completes but in case of non-blocking control return back to the invoking process immediately after invocation, even though the operation has not completed [1].

The objective of the present work is to design a checkpoint algorithm that is suitable for mobile computing environment. Mobile computing environment demands efficient use of the limited wireless bandwidth and the limited resources of mobile machines, such as battery power, memory etc. Therefore in the present work we emphasize on eliminating the overhead of taking temporary checkpoints. To summarize, we have proposed a hierarchical non-blocking checkpointing algorithm in which processes take permanent checkpoints directly without taking temporary checkpoints and whenever a process is busy, the process takes a checkpoint after completing the current procedure.

This paper organized as follows. In section 3 we state the system model considered in this work. In section 4 we have stated the algorithm. In section 5, we have the suitability of our proposed algorithm in the mobile computing environment. Finally section 6 shows the extension of the algorithms.

2. System Model

The system consists of collection of *N* processes, P_1, P_2….P_n, that are connected by channels. There is no globally shared memory and processes communicate solely by passing messages. There is no physical global clock in the system. Message send and receive is asynchronous.

3. Data Structure

Root is the initiator who starts a new consistent checkpoint by taking a tentative checkpoint. All child process take their checkpoint after receiving the checkpoint request (chk_req) message from their parent process, forward request message to its child node and increment to its checkpoint integer number (cin). Each process counts the number of messages it sent and

received in the sr_counter (sent/received counter) variable. Every time a message is sent, the sr_counter is incremented. When a message is received, sr_counter is decremented. When a process receives an chk_tkn request, it adds the sr_counter value from that message to its own sr_counter. When it has received the chk_tkn reply from all its children, it sends the chk_tkn message to its parents. When the root process receives a chk_tkn reply from all its children, and its sr_counter is zero, root broadcasts a commit request (commit_req) message to its children.

When root process receives an update message, it increment in its sr_counter value till the sr_counter value not become zero. When a process receives a commit request it makes its tentative checkpoint permanent and discards the previous permanent checkpoint and propagates the message to its children and wait for the commit acknowledge.

4. Hierarchical Non-blocking Checkpoint Algorithms:

At any instant of time one process act as a checkpoint coordinator called the initiator or root process. Each process maintain one permanent checkpoint, belongs to the most recent consistent checkpoint. During each run of the protocol, each process takes a tentative checkpoint, which replaces the permanent one only if the protocol terminates successfully [6]. In this algorithm if any process is busy with other high priority job, it takes the checkpoint after the job ends. Otherwise it takes a checkpoint immediately. Each process stores one permanent checkpoint. In addition each process can have one tentative checkpoint, and are either discarded or made permanent after some time. Each process maintains a checkpoint integer number (cin), and it is incremented by one in every checkpoint session. Here we use the word checkpoint for tentative checkpoint.

Root process P_i:

There is only one checkpoint initiator or root process which initiates a checkpointing session.
When P_j receives a message from processes P_j, P_k…, P_i takes the tentative checkpoint. After that if it receives any other chk_req it will discard the request.

1. Check direct dependency node ddn_i [] vector.
2. Sends chk_req message to its entire dependent or child processes.
3. Increment in cin_i (cin_i ++).
4. Every time a message is sent, the sr_counter is incremented. When a message is received, sr_counter is decremented.
5. while (sr_counter != 0)
   if receives a chk_tkn response including sr_counter value from all its children it adds the value of sr_counter in its own sr_counter value.
5. if sr_counter = 0
   Send commit_req to all processes to make tentative checkpoint
   permanent and wait for commit_ack.

For Any child processes P_j j! =i and $_1$<=j<= (n-$_1$)

On receipt of checkpoint request:

if P_j receives a checkpoint request
    if P_j has not already participated in checkpoint process
        Take a tentative checkpoint
        Do chkpt_process ()
    else
        If (received cin) > (current cin)   */*Compare both received cin and current cin.*/*
            Take a new tentative checkpoint in place of old one.
            Do chkpt_process ();
        else
            Discard the chk_req and continue normal operation.

On receipt of piggyback application message:

If P_j receives a piggyback application message
    If (received cin > (current cin)   */* Compare both received cin and current cin */*
        Take tentative checkpoint before processing the message.
        Do chkpt
    else
        Ignore the request and continue normal operation.

Procedure chkpt_process ()

If ddn_j[] = = Null   */* for leaf node */*
        Increment in cin_j
        Sends chk_tkn response including sr_counter value to its parent process.
else
        / If $ddn_j[] \neq$ Null */*
        Check ddn_j[] vector.
        Send chk_req to its entire dependent or child node.
        Increment cin_j.
        sr_counter= own sr_counter value + received sr_counter value. */*When Receives chk_tkn response including sr_counter value from its child node*/*
            If receives any update message
                Update sr_counter value and sends this updated message to its Parent process P_i.
            If P_j receives chk_tkn response from all its children processes
                Send chk_tkn response including sr_counter to its parent process P_i .
End procedure

An example

The basic idea of the algorithm is illustrated by the example shown in figure 1. We assume that process P_1 initiates the algorithm. It is also called the root, coordinator or initiator process. First process P_1 takes the tentative checkpoint $Ck_{1,2}$. After that it check its direct dependency node ddn_1[] vector which is { P_1,P_2, P_3}. This means that process P_1 has receive at least one message from P_2, P_3, and P_4. After that P_1 send chk_req to P_2, P_3, P_4 and increment its checkpoint integer number cin 1 to 2 and work as usual. Each time it sends a message, it increase sr_counter and decrease when it receives the message. So in given example sr_counter= -3 which shows that it has received three messages. If sr_counter =0 it meant that it received chk_tkn message from all the processes. Then it sends the commit messages to all its coordinator to convert the tentative checkpoint in to permanent. When it receives the sr_counter from its dependent or child process, it adds this in to its own sr_counter. If it receives any updated message from coordinated or child process it will decrease the sr_counter value and continue this process until or unless the sr_counter ≠ 0. On receiving the chk_req from P_1, process P_2 first take tentative checkpoint $Ck_{2,2}$. After that it check its direct dependent node ddn_2[] vector which is null. It indicates that is a leaf node. So it will take tentative checkpoint and increment in its cin_2 from 1 to 2.

After receiving the chk_req from P_1 process P3 first takes a tentative checkpoint $Ck3_{,2}$ and check its direct dependency node ddn3[] vector which is {P_1, P_5}. Here we are assuming that message $M_{6,2}$ are the late message and process P3 does not receive this message till now. So first process P3 send chk_req message to P_1 and P_5 and after that it increase its checkpoint integer number cin3 from 1 to 2. Similarly process P_4 first take checkpoint $Ck_{4,2}$ and check its ddn_4[] which is {P_6} . Hence P_4 sends a chk_req message to P_6 and increment its cin_4 from 1 to 2. Same process is repeated by the processes P_1 and P_5.

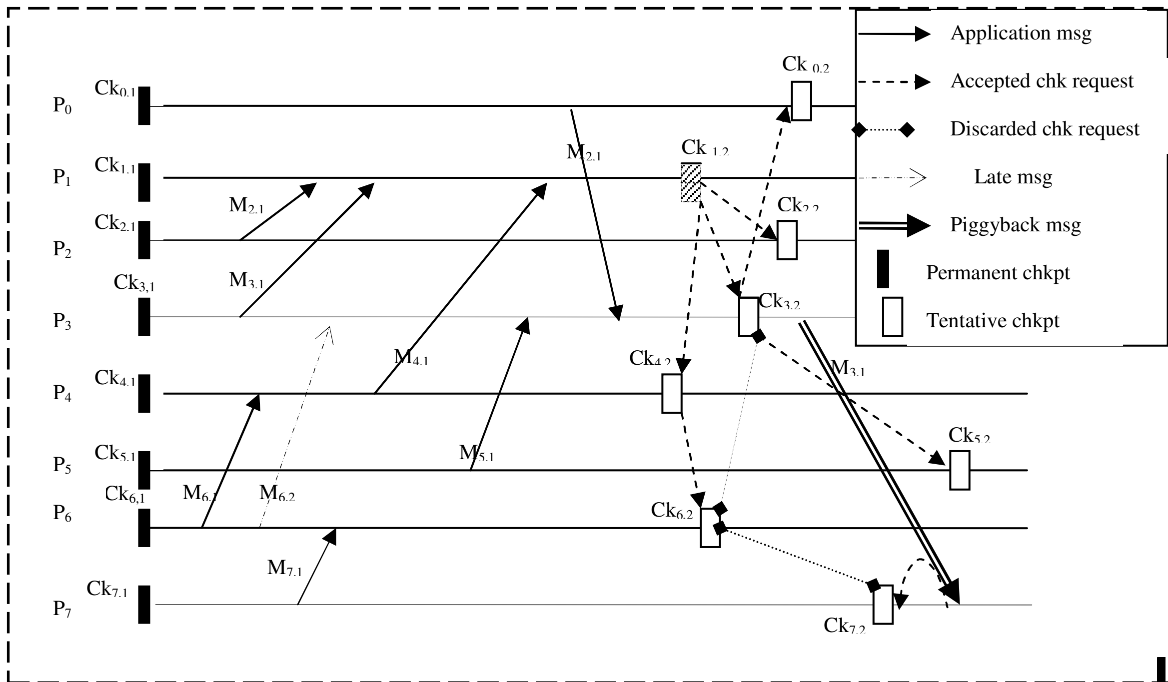

Figure 1: An example of checkpointing approach

Process P_6 receives chk_req from process P_4 first. So it will take checkpoint $Ck_{6,2}$. It is a non blocking checkpointing algorithms. Processes are not blocked after taking checkpoint and free to communicate to other process. Suppose that process P_3 sends an application message $M_{3,1}$ to process P_7. As we know that it is the first application message send by process P_3 after taking its checkpoint $Ck_{3,2}$. So process P_3 send piggyback application message to process P_7 which contain cin value with the message. Now process P_7 compare received cin with current cin which is 1. It finds that received cin 2 is grater than the current cin. So process P_7 takes the checkpoint $Ck_{7,2}$ before processing the message $M_{3,1}$. and increments its cin number from 1 to 2. After that process P_6 receives the message from process P_7. So process P_6 sends a chk_req to process P_7 and increments its cin_6 to 2. It is the second chk_req for process P_7 because it has already taken a checkpoint. In such case process P_7 first compare its current cin_7 with the received cin_6 which is 1. It finds that current cin is greater than the received cin. So it ignores the new checkpoint request.

A leaf process sends chk_tkn message including sr_counter to its parent process after that parent process adds sr_counter in its own sr_counter and when it receives chk_tkn message from all its children it sends to its parent process. This process will be continued until the root process does not receive all messages.

In figure 2 dependency tree sr_counter are shows in brackets. Firstly process P_2 sends its chk_tkn message and sr_counter which is 1 to the root process directly. So the sr_counter of root become -2. Now process P_1, P_5 sends the same to its parent process P_3 receives the same and adds the sr_counter of these processes in its own sr_counter. Now the sr_counter value of the P_3 become 1. As it receive the chk_tkn message and sr_counter value from all its dependent processes. So it sends the chk_tkn message including sr_counter to the initiator process P_1 and P_1 adds the sr_counter in its own sr _counter. Now the sr_counter of initiator process become -1. Then process P_7 sends chk_tkn message including sr_counter which is 1 to its parents process P_6 and after that sr_counter value of P_6 become 2 and then sends the chk_tkn message to process P_4 and after that sr_counter value of process P_4 become 2 and P_4 forward this to the initiator process. Now the sr_counter value of initiator process becomes 1. So root process

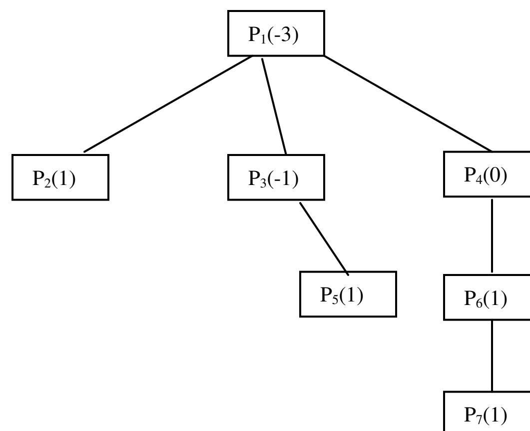

Figure2: Dependency Tree of all processes with sr_counter value before receiving late message send by the process P_6.

receives the chk_tkn message from all the process and its sr_counter value is 1. It shows the inconsistent global state and wait for update message and when the process P_3 receive a message sent by process P_6 its sr_counter value will become -2 and it will send this update message to the root process.

Root process receives the update message from the process P_3 and decrement its sr_counter by 1. So now the sr_counter value of root process become 0(zero) .Root process send the commit_req to the entire child node. When a process receives a commit_req message, it makes its tentative checkpoint permanent and discards the previous permanent checkpoint.
On the other side when the process P_6 receive the chk_req sent by process P_3 it compare its current checkpoint integer number cin_5 with the received checkpoint integer number cin_3. It finds that current cin_6 2 is greater than the received cin_3 which is 1. So it discards the request.

5. Suitability for Mobile Computing Environment

Consider a distributed mobile computing environment. In such an environment, only limited wireless bandwidth for communication among the computing processes. Besides, the mobile hosts have limited battery power and limited memory. Therefore, it is required that, any distributed application running. It is required that, any mobile distributed application running in such an environment must make efficient use of the limited wireless bandwidth, and mobile hosts' limited battery power and memory. Below we show that the proposed algorithm satisfies all the above three requirements.

a) This algorithm, processes neither take any useless and unnecessary checkpoints which help in better utilization of the mobile host limited memory.
b) This algorithm uses the minimum number of control messages. It definitely offers much better bandwidth utilization.

6. Extension of the Algorithms

The algorithms so far discussed, considers that there is only one checkpoint initiator. In case there are multiple concurrent initiators, each process has to handle multiple checkpoint sessions concurrently, and also maintain synchronization among them. A comparative study can also be done with other existing algorithms.

Reference:

[1] *Kshemkalyanl Ajay D, Singhal, M.: Distributed Computing Principals, Algorithms, and System*

[2] Singhal, M. , Shivaratri, N.-G.: Advanced Concept in Operating System. McGraw Hill,(1994)

[3] Cao, G. and, Singhal, M "Mutable checkpoints: a new checkpointing approach for mobile computing systems,"IEEE Transactions on Parallel and Distributed Systems, vol. 12, Issue 2,pp. 157-172, Feb 2001.

[4] Coulouris, G., Dollimore, J., Kindberg, T., Distributed System Concepts and Design, 3rd edition, Addison- Weslely,(2001), 772p

[5] Elnozahy,E.N, Johnson, D.B. and Zwaenepoel, W. "The Performance of Consistent" Proceedings of 11th Symp. On Reliable Distributed Systems, pp. 86-95, October 1992, Houston.

[6] Koo R. and Toueg. S, "Checkpointing and Rollback-Recovery for distributed System," IEEE Trans. Software Eng., SE-13(1):23-31, January 1987.

[7] Ziv Avi and Bruck Jehoshua "Checkpointing in Parallel and Distributed Systems", Book Chapter from Parallel and Distributed Computing Handbook edited by Albert Z. H. Zomaya, pp. 274-320, Mc Graw Hill, 1996.

[8] Bhargava B. and Lian S.R., "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed System -An Optimistic Approach," Proceeding of 17th IEEE Symposium on Reliable Distributed System, p. 3-12, 1988.

[9] Chandy K.M. and Lamport L., "Distributed Snapshots: Determining Global State of Distirbuted Systems,"ACM Transaction on Computing Systems, vol. 3 No. 1, pp. 63-75, Feb. 1985.

[10] Elnozahy E.N., Alvisi L., wang Y.M. and Johnson D.B., "The Performance of Consistent Checkpointing," Proceedings of the 11th Symposium on Reliable Distributed Systems, pp. 39-47, October 1992.

[11] Koo R. and Toueg S., "Checkpointing and Roll-Back Recovery for Distributed System," IEEE Trans.on Software Engineering, vol. 13, no. 1, pp. 23-31, January 1987.

[12] Randall, B, " System structure for Software Fault Tolerance", IEEE Trans.on Software Engineering, Vol.1,No.2,pp220-232, 1975.

[13] Russell, D.L., "State Restoration in System of Communication Processes", IEEE Trans. Software Engineering, Vol.6,No.2pp 183-194, 1992.

[14] Sistla,A.P. and Welch,J.L., "Optimistic Recovery in Distributed Systems", ACM Trans. Computer System, Aug, 1985, pp. 204-226.

[15] Wood, W.G., " A Decentralized recovery Control Protocol", IEEE Symposium on Fault Tolerant Computing. 1981.

[16] Gupta Bidyut .el "A low-Overhead Non block Checkpointing Algorithm for Mobile Computing Environment" springer-Verlag Berlin Heidelberg 2006 pp. 597-608.