MESSAGE INDUCED SOFT CHEKPOINTING FOR RECOVERY IN MOBILE ENVIRONMENTS

Ruchi Tuli[1] & Parveen Kumar[2]

[1]Research Scholar, Singhania University, Pacheri Bari (Rajasthan) India
[2]Professor, Meerut Institute of Engineering & Technology, Meerut (INDIA)

Abstract

Checkpointing is one of the commonly used techniques to provide fault tolerance in distributed systems so that the system can operate even if one or more components have failed. However, mobile computing systems are constrained by low bandwidth, mobility, lack of stable storage, frequent disconnections and limited battery life. Hence checkpointing protocols which have fewer checkpoints are preferred in mobile environment. Since MHs are prone to failure, so they have to transfer a large amount of checkpoint data and control information to its local MSS which increases bandwidth overhead. In this paper we propose a new soft checkpointing scheme for mobile computing systems. These soft checkpoints are saved locally in the host. Locally stored checkpoints do not consume network bandwidth and can be created in less time. Before disconnecting from the MSS, these soft checkpoints are converted to hard checkpoints and are sent to MSSs stable storage. In this way, taking a soft checkpoint avoids overhead transferring large amount of data to the stable storage MSSs.

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

1. Introduction

As computer and wireless communication technology advance, the paradigm of mobile computing becomes close to reality. Mobile users are able to access and exchange information while they are on the move. As a result, collaborative work can be done easily, no matter where the participating members are physically located. Moreover, since user mobility is supported, it is possible that joint workers are distributed over the wide area network and each of them is connected to the network via a wireless link. For example, in a sensor network which carries out a real-time scientific computation, sensors with processing capability can be mobile and distributed. Traveling salespersons may rely on the gradation of each other for making an appropriate decision at a particular time. In these scenarios, the most important thing is to make sure their work is progressing, and to minimize the lost work if a failure occurs. To achieve high reliability, checkpointing and rollback recovery techniques are widely used in the parallel and distributed computing environment. Recently, checkpointing protocols for mobile computing systems have also been proposed[1]. A common goal of checkpointing protocols for mobile environment is to avoid extra coordination messages and unnecessary checkpoints. Coordinated checkpointing protocols have the advantage of simplicity over uncoordinated checkpointing protocols in terms of recovery and garbage collection. Besides, the former requires less storage capacity for saving checkpoints.

In this paper, we focus on checkpoint based recovery technique for mobile computing systems. A checkpoint protocol typically functions as follows : the protocol periodically saves the state of the application on stable storage. When a failure occurs, the application rolls back to the last saved state and then restarts its execution. Checkpoint protocols proposed in the literature are not suitable for mobile environments because of disconnections.

2. Related work and problem formulation

2.1 Related Work

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

consistent global checkpoint, processes must synchronize their checkpointing activities. In other words, when a process takes a checkpoint, it asks to all relevant processes by sending checkpoint requests to take checkpoints. Therefore, coordinated checkpointing suffers from high overhead. The protocol presented in this paper shows performance improvement over the work reported in [3], [4], [6], [7] & [8]. The protocol designed by Acharya and Badrinath [6] requires to create a new checkpoint whenever they receive a message after sending a message. Processes also have to create a checkpoint prior to disconnection. Pardhan et al. [7] proposed two uncoordinated protocols. The first protocol creates a checkpoint every time when a process receives a message. The second protocol creates checkpoints periodically and logs all messages received.

The protocols proposed in [3], [4] & [8] follow two-phase commit distributed structure. In the first phase processes take temporary checkpoints when they receive the checkpoint request. These tentative checkpoints are stored in stable storage of MSS. In the second phase, if an MSS learns that all the processes have taken the temporary checkpoints successfully, initiator MSS sends commit message to all the participating nodes. In these checkpoints an MH has to transfer a large amount of data to its local MSS over its wireless network which results in high checkpoint latency and recovery time as transferring such temporary checkpoints on stable storage may waste a large amount of computation power, bandwidth, energy and time. The protocol proposed by us creates a checkpoint whenever the local timer expires, and it only logs the unacknowledged messages at checkpoint time. Our protocol uses two types of checkpoints to recover from failure. The two previous protocols proposed in [6] and [7] always assume hard failures.

2.2 Problem formulation

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

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

2.3 Basic idea

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

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

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

2. The proposed checkpointing Algorithm

3.1 System Model

The mobile environment model used in this protocol contains both fixed and mobile hosts interconnected by a backbone network. The fixed hosts are called MSS and mobile hosts are connected to MSS by wireless links. A

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

3.2 Algorithm Concept

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

3.3 Creation of a global state

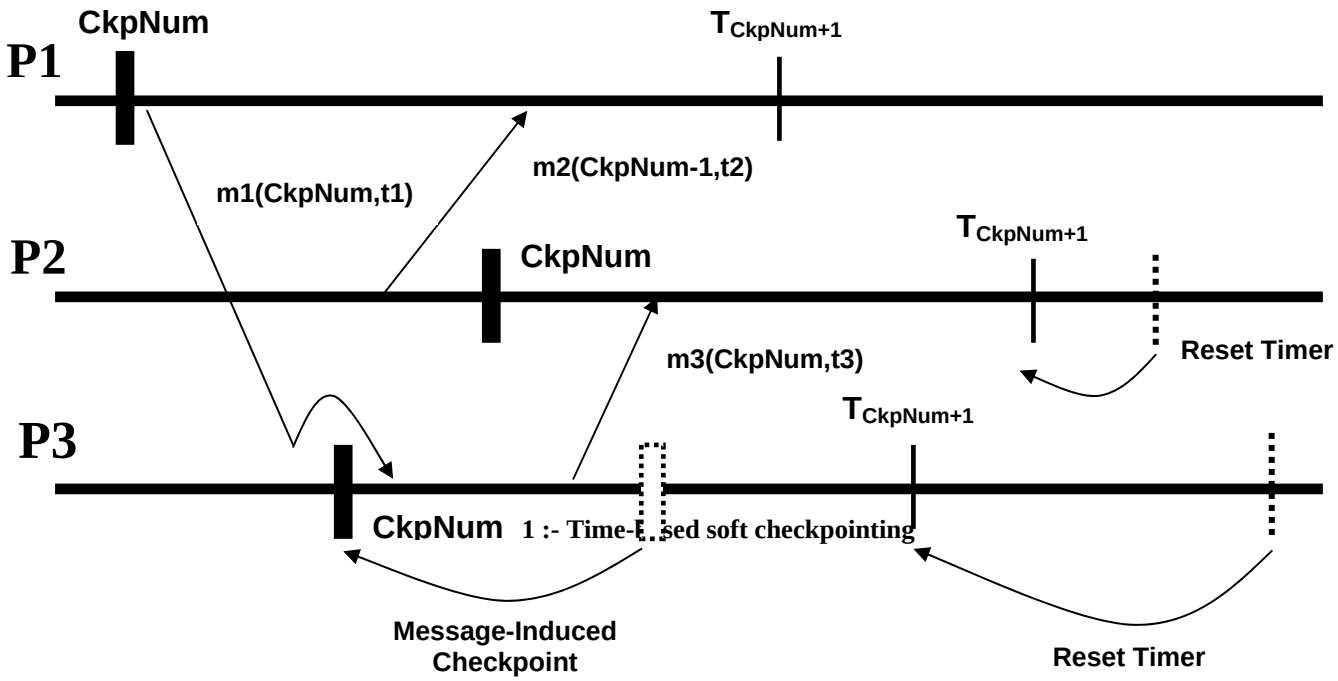
Whenever a mobile host moves out of the range of a cell or user turns off the network interface, it becomes disconnected. In a disconnected mode, the mobile host cannot access any information that is stored on a stable storage. Due to this reason, the protocol must be able to perform its duties correctly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

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

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

3.4 Working of the Algorithm

We illustrate the execution of the protocol with the help of following figure. This figure (Figure 1) represents the execution of three processes. Processes create their checkpoints at different instants, because timers are not synchronized. After saving its *CkpNum* checkpoint, process P1 sends message *m1*. When m1 arrives, process P3 is still in its *CkpNum-1* checkpoint interval. To avoid a consistency problem, P3 first creates its *CkpNum* checkpoint, and then delivers *m1*. P3 also resets the timer for the next checkpoint. Message *m2* is an in-transit message that has not been acknowledged when process P2 saves its *CkpNum* checkpoint. This message is logged in the checkpoint of P2. Message *m3* is a normal message that indirectly resynchronizes the timer of process P2. It is possible to observe in the figure the effectiveness of the resynchronization mechanism.



3.5 The Algorithm

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

// S_i - Sender's Identifier
// $CkpNum_i$ – Current checkpoint number of the sender
// $timeToCkp_i$ – Time interval until next checkpoint
// msg_i – Message contents

A . Message Receiving

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

B. Application Process (At MH)

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

C. Stable Storage (At MSS)

// The function arguments are same as in message receiving

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

```
        CkpHard : = CkpNum_i;
        garbageCollect (CkpHard);
}
```

The functions given above are used to create a new checkpoint. Function *createCkp* is called to save a new process state. It starts by incrementing the *CkpNum*, and then it resets the timer with the checkpoint period. Next, the function determines if the checkpoint should be saved locally or sent to stable storage. The function *saveState* stores the process state locally, and the function *sendCkp* sends the process state to stable storage. The function *receiveCkp* is called by the stable storage to store newly arrived checkpoints. It first writes the received state to the disk, and then updates the local checkpoint counter. Then, it determines if a new hard global state has been stored using a checkpoint table. The checkpoint table contains one row per *CkpNum*, and one column per process. The table entries are initialized to zero. An entry is set to one whenever the corresponding checkpoint is written to disk. The table only needs to keep one bit per entry, which means that it can be stored compactly. A new hard global state has been saved when all entries of a row are equal to one. The variable *CkpHard* keeps the checkpoint number of the new hard global state. The function *garbageCollect* removes all checkpoints with checkpoint numbers smaller than *CkpHard*.

3.6 Proof of Correctness

Theorem *1* The algorithm ensures a consistent global checkpoint

Proof The consistency property is ensured if no process receives a message $CkpNum_i$ larger than current *CkpNum*. The process creates a new checkpoint before delivering the message to the application if $CkpNum_i$ is larger than the local *CkpNum*. All processes restart from an appropriate state because, if they decide to restart, they resume execution from a consistent state (the checkpointing algorithm takes a consistent set of checkpoints). We consider following three cases :-

Case 1: For all MHs about to disconnect, the algorithm takes a checkpoint before the disconnection. Hence, its checkpoint is used as an initial checkpoint after a reconnection. Therefore, no inconsistent state occurs due to the disconnection.

Case 2: For all MHs in doze mode, the algorithm sends a request message to wake up them to take a checkpoint. Hence, all MHs in doze mode can not cause an inconsistent state.

Case 3: When a MH moves from a cell of MSS_i to another cell of MSS_j, MSS_j sends a hand-off message to MSS_i to obtain the MH's checkpoints. Hence, a consistent state is kept when a MH moves.

As a discussion above, we prove that the algorithm ensures a consistent global checkpoint even if it is a special case.

Theorem 2 The algorithm rolls back all necessary processes to a global consistent state when failure occurs.

Proof An orphaned message can cause an inconsistent state after a failure recovery. An orphan message M occurs when the states of MH_i and MH_j are rolled back to checkpoints $C_{i,a}$ and $C_{j,b}$, where a and b are checkpoint indices of MH_i and MH_j, and M is sent after $C_{i,a}$, but it is received before $C_{j,b}$. There are two cases depending on whether the message is sent before or after the failure. If it occurs before the failure, MH_i sends message M to MH_j after $C_{i,a}$, and MH_j receives M, and then takes $C_{j,b}$. Based on Section 3.4 part A we prove that recoverability property is guaranteed by logging at the sender all messages that might become in-transit. These are the messages that have not been acknowledged by the receivers at checkpoint time. The sender process also logs the send and receive sequence number counters. During normal operation, these counters are used by the communication layer to detect lost messages and duplicate messages due to retransmissions. After a failure, each process resends the logged messages. Duplicate messages are detected as they are during the normal operation

4. Conclusion

In our proposed approach, a have described a protocol that is able to save consistent recoverable global states. The process creates a new checkpoint whenever the local timer expires. The protocol stakes a soft checkpoint and saves it on the mobile host and later on before disconnection converts it to the hard checkpoint that is stored on MSS as soft checkpoint is less reliable. When the mobile host is disconnected, the protocol creates soft

checkpoints to recover from soft failures. The main features of our algorithm are: (1) it is non-blocking; (2) it is adaptive because it takes checkpointing decision on the basis of checkpoint sequence number.

REFERENCES

[1] C. Chowdhury, S. Neogy, "A Consistent Checkpointing- Recovery Protocol for Minimal number of Nodes in Mobile Computing System", *in International Conference on High Performance Computing*, pp-599-611, 2007.

[2] Koo, R. and Toueg, S. (1987) 'Checkpointing and roll-back recovery for distributed systems', IEEE Trans. on Software Engineering, Vol. 13, No. I, January, pp.23–31.

[3] G. Cao and M. Singhal, "On Coordinated Checkpointing in Distributed Systems," IEEE Trans. Parallel and Distributed System pp. 1213-1225, Dec. 1998.

[4] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel, ªThe Performance of Consistent Checkpointing,º Proc. 11th Symp. Reliable Distributed Systems, pp. 86-95, Oct. 1992.

[5] R. Prakash and M. Singhal, "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems," IEEE Trans. Parallel and Distributed Systems, pp. 1035-1048, Oct. 1996.

[6] Acharya, A. and Badrinath, B.R. Checkpointing distributed applications on mobile computers. In Proceedings of the Third International Conference on Parallel and Distributed Information Systems (Austin, Texas, Sep, 1994), pp 73-80.

[7] Pradhan, D.K., Krishna, P., and Vaidya, N.H. Recovery in mobile environments: Design and trade-off analysis. In Proceedings of the 26th International Symposium on Fault-Tolerant Computing, (Sendai, Japan, June 1996), IEEE, pp. 16–25.

[8] G. Cao and M. Singhal., "Mutable Checkpoints : A New checkpointing Approach for Mobile Computing Systems", In Proceedings of the IEEE Trans. Vol. 12, No. 2, pp-157-172, Feb. 2001

[9] Randell, B. System structure for software fault tolerance. IEEE Trans. Softw. Eng. SE-1, 2 (June 1975), 220–232.

[10] N. Neves, "Time-based coordinated checkpointing," Ph.D. dissertation, UIUCDCS-R-98-2054, University of Illinois at Urbana-Champaign, 1998.

[11] Parveen Kumar, R K Chauhan, "A Coordinated Checkpointing Protocol for Mobile Computing Systems", International Journal of Information and Computing Science, Vol. 9, No. 1, pp. 18-27, 2006.