A Checkpointing Algorithm for Mobile Computing Environment

Pushpendra Singh and Gilbert Cabillic

Aces Projet, INRIA-Rennes,
Campus Universitaire de Beaulieu
35042 Rennes, France

{Pushpendra.Singh,Gilbert.Cabillic}@irisa.fr

Abstract. Mobile computing is going to change the way, computers are used today. However mobile computing environment has features like high mobility, frequent disconnections, and lack of resources, such as memory and battery power. Such features make applications, running on mobile devices, more susceptible to faults. Checkpointing is a major technique to confine faults and restart applications faster. In this paper, we present a coordinated checkpointing algorithm for deterministic applications. We are using anti-messages along-with selective logging to achieve faster recovery and reduced energy consumption. Our algorithm is non-blocking in nature and avoids unnecessary computation. We ask only minimum number of processes to take the checkpoint and also take in to account the limited storage available at mobile devices.

1 Introduction

Mobile environment has features like high mobility, frequent disconnections, and lack of resources such as memory and battery power; such features make applications, running on mobile devices, more susceptible to faults. Thus providing good fault-tolerant mechanisms is necessary to ensure reliable working of computing devices. Coordinated Checkpointing is a major technique to provide fault-tolerance.

In this paper, we present a coordinated checkpointing algorithm for deterministic applications. Coordinated checkpointing has two phases. In the first phase, checkpoint initiator asks other processes to take a checkpoint. If every process takes a checkpoint, then the initiator enters into the second phase and asks processes to make their recent checkpoint permanent. However, in the first phase, if a process denies to take a checkpoint then whole checkpointing process is aborted. In coordinated checkpointing, last checkpoints of every process form a consistent state; thus it is free from domino effect [1] and problem of storing number of checkpoints.

Recently proposed checkpointing algorithms [2,3] do not take into account the nature of application and insist on avoiding orphan messages [3] to have a correct recovery. We have found out that for deterministic applications, their

Fig. 1. Assumption of same processing

approach results in repeated computation and slow recovery. We have developed our algorithm only for deterministic applications and orphan messages are present in the checkpointing states of processes in our algorithm. However, we show that we have a faster and more energy-aware recovery than [2,3]. We have used anti-messages [4] to make our algorithm non-blocking; but, we store anti-messages only at the receiver's end, and only for messages that can become orphan [3], which we call, *selective logging*. In our algorithm, only the processes dependent on each other take checkpoints and not every process; a process i is dependent on process j , if it has received at least one message from process j since its last checkpoint [5].

For deterministic applications, we assume the model of Johnson-Zwaenepoel [6] i.e. if two processes start in the same state, and both receive the identical sequence of inputs, they will produce the identical sequence outputs and will finish in the same state. The state of a process is thus completely determined by its starting state and by the sequence of messages it has received. For example in Fig. 1, suppose P_1 has a permanent checkpoint before sending m_1 to P_2 and after sending of m_1 some fault occurs at P_1 . So when P_1 restarts from its last checkpoint, it will send m_1 again to P_2 .

The rest of the paper is organized as follows: section 2 provides necessary background. In section 3, we present our checkpointing algorithm. An analysis of algorithm and comparison with other works is done in section 4. We give our future direction and conclude in section 5.

2 Preliminaries

1. *System Model:* Mobile Computing environment consists of fixed base stations, known as mobile service stations (MSS), and mobile hosts (MH). A MSS is responsible for MHs present in its cell. Communication between MH and its corresponding MSS is through reliable *FIFO* wireless channel. A MH can directly talk only to the MSS responsible for it. Communication between a MH and another MH or MSS is only through the MSS responsible for that MH. MSSs are connected together through a fixed wired network, communicating through a reliable *FIFO* wired channel.

Our distributed computation model consists n processes $\{P_1, P_2, \dots, P_n\}$. The processes run on same (or different) MH(s) or MSS(s) with fail-stop computation model. Processes do not share a global memory or global clock. However every process has access to the stable memory of its MH or MSS.

Data stored on stable storage is shielded from failures. The processes interact with each other by message passing and no other means. Delays in communication channels are arbitrary but finite.

2. *Anti-message*: When a process receives a message, it can store an anti-message for it. An anti-message [4] is same as the original message. The function of anti-message is to annihilate duplicate original message, if sent again. For example in Fig. 1, P_2 can save an anti-message for m_1 , when m_1 was sent first time. Now, if P_1 restarts, after a fault, and sends m_1 again, because of deterministic nature, then P_2 can delete m_1 with its anti-message.
3. *Selective Logging* : Our logging is selective in nature because we do logging only during the checkpointing's phase one and phase two and only for possible orphan messages, as explained later in section 4. This logging is very much limited in nature in comparison to [6], where every message is logged, since last checkpoint.

3 The Checkpointing Algorithm for Deterministic Applications

To explain our approach clearly, we assume only one checkpoint initiator at a single time.

3.1 The Basic Idea

The basic idea of algorithm is as follows: in the first phase, a process starts checkpointing as checkpoint initiator by sending requests to processes over which it depends. When a process receives a checkpoint request, it takes the checkpoint, propagates the checkpoint request and continues its processing. During the checkpointing, when P_i sends a computation message to P_j , it also sends id of current checkpoint initiator, if P_i has taken a checkpoint, otherwise a NULL value. When P_j receives the message, it checks the value of checkpoint initiator. If it is not NULL then P_j knows that checkpointing is going on and it might receive a checkpoint request later; thus it saves an anti-message for the received message, before processing it. However if value of checkpoint initiator is NULL, it simply processes the message. After a process has taken the checkpoint, whenever it receives a computation message from another process, it stores it (before processing) in a serial order to previously received such messages. An anti-message is also stored for same message. This storing is done only during the checkpointing to insure same processing in the recovery phase. In the second phase, checkpoint initiator asks processes to make their checkpoint permanent. The checkpoints stored in first phase are made permanent and checkpointing process ends.

The saved anti-messages, messages and checkpoints form the complete package for recovery. If a fault occurs, processes start from their last checkpoint and executes the messages, they had stored during checkpointing, in the same serial order as they were received. If during the course of recovery a process receives

Fig. 2. Existing Problems

a message for which it has an anti-message, it annihilates the message without processing it.

3.2 An Improved Version

In our basic scheme, processes might keep storing unnecessary messages and anti-messages. For example in Fig. 2(a), if P_1 later takes another checkpoint CP_2^1 , storing of anti-message of m_1 at P_2 becomes unnecessary because now P_1 will not rollback to CP_1^1 .

We solve this problem by passing the information of last checkpoint to other processes through a variable *cp_number*. Every process has a variable *cp_number*, denoted by cp_number_i for P_i . The cp_number_i contains the number of last permanent checkpoint of P_i . When a process sends another process a computation message, it will also send value of its *cp_number*. When P_j receives the message from P_i , it checks the value of cp_number_i . If P_j has stored anti-messages for sender P_i , with a value of cp_number_i less than newly received cp_number_i , it will delete all such anti-messages.

Besides it, in our algorithm, a process asks other processes to take the checkpoint based on dependency information [5]. But this dependency information may be outdated, thus a process can be forced to take unnecessary checkpoints [2]. For example in Fig. 2(b), if P_2 decides to take CP_2^2 , it will ask P_1 to take a checkpoint based on the dependency caused by message m_1 . However this dependency information is outdated because P_1 has taken other checkpoints CP_2^1 and CP_3^1 , which removes the old dependency caused by m_1 .

To solve this problem, every process has a boolean array *message_sent*, denoted by $message_sent_i$ for P_i ; which is initialized to zero. When P_i sends a message to P_j since its last checkpoint state, it will set value of $message_sent_i[j]$ to 1. $message_sent_i$ will be reset to zero after a successful checkpoint of P_i . So when P_j receives a checkpoint request from P_i , it will check the value of $message_sent_j[i]$. If the value is 1, it will take the checkpoint and will propagate the checkpointing request, otherwise, it will ignore the request and will send P_i a *cp_ignore* message. On receiving a *cp_ignore* message, P_i will not include P_j in second phase of checkpointing.

Cao-Singhal [2] have tried to solve the same problem using an array of *checkpointing sequence number (csn)*, which is same as our *cp_number*. However to maintain the information of *csn*, they need an integer array, which requires more storage than our boolean array; besides, *message_sent* is also used in reducing the number of system messages as explained later.

Data Structures

1. checkpoint Number: A process gives a unique serial number to the checkpoint taken by it i.e. i^{th} checkpoint taken by P_k is denoted as CP_i^k . After a successful checkpoint, checkpoint number is stored permanently at a process.
2. cp_finish: It will be used to find out the end of first phase of checkpointing. We will use following technique to know the completion: every process will have a variable *cp_finish*. A process will increase *cp_finish* by 1 with every checkpoint request sent and will decrease by 1 with every checkpoint request received. When the process has taken its checkpoint it will send value of its *cp_finish* to the initiator. Initiator will add all the *cp_finish*, received by it, with its own *cp_finish*. When the sum will become 0, it will understand that every process has taken the checkpoint and it will start the second phase.
3. dependency array: Every process has a n-bit array, which is initialized to zero [5]. When P_i receives a message from P_j , since its last checkpoint, it sets j^{th} bit of its dependency array to 1. *Dependency array* is again reinitialized to zero when a process completes a successful checkpoint. This array is used to maintain the information of dependency between processes, so that only processes dependent on each other take checkpoints.
4. message_sent array: Every process has a n-bit array, which is initialized to zero. *message_sent* array will be used to remove unnecessary checkpoint requests as explained earlier and also in reducing number of system messages as follows: a logical AND operation on *dependency array* and *message_sent* array will give the number of checkpoint requests a process will receive during a checkpointing process. So when a process receives first checkpoint request, it can calculate the final value of *cp_finish* and send it to the checkpoint initiator. Later when it receives other checkpoint requests, it can simply ignore them without notifying checkpoint initiator. Without *message_sent* array a process would have had to notify the initiator in response of every checkpoint request, as it is done in [5], otherwise *cp_finish* would not add up to 0.
5. cp_initiator: Every process has a field of *cp_initiator*, denoted by *cp_initiator_k* for P_k . This field contains *id* of the checkpoint initiator, while checkpointing is going on, otherwise this field is set to NULL.

Algorithm: To explain our approach clearly, we assume only one checkpoint initiator at a single time, however our approach can be extended to multiple initiators by extending the data structures appropriately.

1. Our distributed computation model consists n processes $\{P_1, P_2, \dots, P_n\}$.
2. Initiator will take a checkpoint and then it will send checkpoint requests to the processes, over which it depends using its dependency array. It also sends its dependency array, its *id* as initiator, and its *cp_number*.
3. When a process receives a checkpoint request, it will check the validity of request, using its *message_sent*. If the request is valid, it will take a checkpoint, will propagate the checkpointing request, and will continue its normal processing. It will store the value of checkpoint initiator in its *cp_initiator*

field so that later if it receives a duplicate request, it will discard the request. If the request is not valid then it will discard the request and will notify the initiator.

4. When a process propagates a checkpointing request, it sends the request only to the processes over which it depends and they have not already got the request from initiator. This information is obtained by doing a logical AND operation on the dependency array, received with checkpoint request, and its own dependency array. Besides, when process sends new checkpoint request it will send total number of processes, for whom request has been sent by doing a logical OR operation on dependency array received in checkpoint request and its own dependency array [2]. It will also send the value of *cp_initiator*.
5. If a process receives a computation message, it will do one of the two things:
 - (a) If the process had previously received a checkpoint request and have taken a checkpoint then it will store the message with *cp_number* and *id* in a serial order to previous similar messages, before processing. An anti-message will also be stored for this message.
 - (b) If the process is not aware of any checkpointing process going on then it will check the checkpoint initiator field of the message.
 - i. If the value of the checkpoint initiator field is NULL, it will simply process the message.
 - ii. Otherwise, it will store an anti-message with *cp_number* and *id* for the message before processing it
6. After taking a checkpoint in the first phase, every process will send the initiator a checkpoint completion message alongwith its *cp_finish*.
7. Initiator will add all *cp_finish*, received by it, to its own and when its *cp_finish* will become 0, it will start the second phase.
8. In the second phase, initiator will ask participating processes to make their checkpoints permanent and will make its own checkpoint permanent.
9. Databases like dependency array, message_sent array, etc., will be reinitialized to 0. New permanent checkpoint numbers will be assigned.
10. This will end the checkpointing process.
11. When a process receives a computation message, containing higher *cp_number* of a sender, for which the process has stored anti-messages or message with a less *cp_number*; it will delete all such anti-messages.

Recovery: For recovering from failure, every process will begin from its last stored checkpoint state. A process will do its processing as it was doing before failure. If the process had stored some messages, it would process them in the same serial order as they had arrived. When a process receives a message for which it has an anti-message, it will delete the message with its anti-message without processing.

Fig. 3. Example

3.3 Example

In Fig. 3(a), P_2 is the initiator and it depends upon P_3 and P_4 because it has received messages from them in past. P_2 starts checkpointing and sends checkpoint requests to P_3 and P_4 and increases cp_finish_2 by 2. P_4 receives the request, takes a checkpoint, decreases its cp_finish by 1 (because it has received one request but sent none), sends it to P_2 and continues its processing. P_4 sends message m_1 to P_3 with checkpoint initiator id equal to 2. When P_3 receives m_1 , it finds a not NULL value for checkpoint initiator. It stores the anti-message for m_1 before processing and continues its processing. P_3 sends m_2 with checkpoint initiator field NULL. P_1 processes m_2 . P_3 receives checkpoint request, takes a checkpoint, sends a checkpoint request to P_1 over which it depends. Its cp_finish remains 0 (because it has received one request and sent one), which it sends to P_2 . P_1 sends m_3 to P_3 with checkpoint initiator field NULL. Since P_3 have taken a checkpoint, it stores m_3 , giving it a serial number; it also stores an anti-message for m_3 before processing it. Meanwhile checkpoint requests reach P_1 , it takes the checkpoint, decreases its cp_finish by 1 and sends it to P_2 . P_2 adds all the cp_finish 's received by it to its own and as it becomes 0, P_2 enters into the second phase and asks every process to commit their checkpoints and completes the checkpointing. Checkpoints CP_1^1 , CP_1^2 , CP_1^3 , CP_1^4 , message m_3 , and anti-messages of m_1 and m_3 form complete recovery package.

If some fault occurs, system starts from last saved checkpoints i.e. CP_1^1 , CP_1^2 , CP_1^3 , and CP_1^4 . P_4 sends message m_1 to P_3 as it has done before failure. When P_3 receives m_1 , it annihilates m_1 . Later P_3 process m_3 from its log and system progresses.

Here we see that P_3 annihilates m_1 (which is an *orphan message*), without processing, thus hiding its presence. In other words we have basically shifted the checkpoint state of P_4 to after the sending of m_1 , as shown in Fig. 3(b); which is a complete consistent state. Thus we recovered as if there was no *orphan message*.

Later, if P_4 takes another checkpoint CP_2^4 and sends a message m_4 to P_3 , P_3 will delete the anti-message of m_1 . Similarly stored message m_3 and its anti-

message will also be deleted when P_3 will take CP_2^3 or when it will receive a message containing a larger checkpoint number of P_1 than 1.

3.4 Advanced Recovery

We can enhance our recovery to a more energy-aware one by using following scheme: every message will have two parts viz. *header* and *content*. In the beginning only *header* will be sent to the receiver, which will also contain total size of *content* with usual routing information. If a process receives a message header for which it has an anti-message, it will do one of the two things:

1. Continue receiving it and later delete it with its anti-message.
2. Depending on the size of content, if the receiver thinks that it is a waste to receive the message just to be deleted later, for example for big messages. It will send “not to send” message to the sender for this message.

When a sender receives a “not to send” message from the receiver, it will not send the *content*. This enhancement involves use of an extra message viz “not to send”. Generally extra messages are advised to be avoided given the argument that they involve extra energy consumption. However receiving and sending of a message, which is to be discarded later, also involves consumption of energy. For big messages say 1 Mb, this energy consumption can be more than energy consumed in sending one extra message. Besides it “not to send” message is very lightweight compared to big size computation messages, thus a flexible approach can be adopted by receiver, if it thinks that it will be wise to send an extra message, it will do so, otherwise it will simply receive the message and discard it.

4 Analysis of Algorithm and Comparison with Other Works

We have made our algorithm for deterministic application to take advantage of the nature of application in saving resources such as energy and memory of a mobile device. In this section we will analyze our algorithm for similar reasons.

1. Storage : As a part of our algorithms, we store checkpoints and do some message logging. But this message logging is done only for a brief period i.e. between phase 1 and phase 2 of the checkpoint algorithm and only when the sending process has not taken a checkpoint but receiving process has taken a checkpoint. We store anti-messages only for the messages that can become orphan. Anti-messages are not stored for the messages, where receiver has taken the checkpoint or sender has not taken the checkpoint like in Fig. 3, anti-message is stored only for m_1 and not for m_2 . As mentioned in our enhanced recovery scheme, for anti-messages we can store only the message header, which will not consume much of storage.

Unnecessary anti-messages are also deleted as part of algorithm. Since our algorithm is based on coordinated checkpointing scheme, so only last checkpoint is needed to be stored.

Previous work has been done on the same approach by [7,6]. However, they log every message since last checkpoint and also block the processing till logging is not done for a message. The overload of such logging is very high for mobile devices because of limited storage available.

2. Energy : The algorithm saves energy by avoiding the same calculation that it has already done. For example in Fig. 3, m_1 is an orphan message. Other checkpointing algorithms [8,9,2] avoid orphan message by taking checkpoint prior to processing the message. Which basically means that at the time of recovery such messages are again processed by receiver, because they had not been included in the checkpoint. This repeated execution of same messages consumes energy.

In our approach receiver will discard such messages, thus we save energy by avoiding the same computation unlike [8,9,2]. Besides it, with our enhanced recovery scheme, whole sending and receiving of a big message can be avoided by using an extra lightweight message. It will result in energy saving by avoiding the communication as well.

3. Execution time : The arguments of energy saving also holds true for execution time for deterministic recovery. Since a receiving process does not compute the same messages, thus has less average time for other messages; this results in faster recovery. Again taking example of Fig. 3, suppose a fault occurs after the reception of m_3 . During the recovery, our algorithm would not compute m_1 and other such messages, unlike [8,9,2].

Recently developed algorithms [8,9,2] do not take advantage of deterministic nature of application and applies same method for every kind of application. We think that such approach is not good for small wireless devices, such as PDA's, working in mobile environment. Because for such devices, even small savings of energy and storage are substantial; thus we advocate use of different techniques suitable to the natures of applications to achieve different gains [10].

5 Conclusion and Future Work

In this paper, we presented a checkpointing algorithm suitable to deterministic application. We used anti-message technique [4] with selective logging to achieve our aim. With the help of anti-messages, our approach avoids same computation during the recovery phase; so it provides a faster recovery and saves energy. We have limited our logging to a small period thus without causing extra overhead on mobile device for storage. We find that our algorithm results in longer utilization of device by conserving energy then previously proposed algorithms [2, 11]. However our algorithm is limited to the deterministic applications, which we find a limitation in evolving mobile computing field.

Presently we are working on developing an algorithm for non-deterministic applications. In our approach, we are using *successive checkpoint* approach to

achieve our aim [12]. *Successive checkpoint* is an extra checkpoint taken to avoid orphan message. However our approach is totally different than taken by [2], because our successive checkpoint will never be discarded unlike their *mutable checkpoint*. Thus taking a successive checkpoint will not result in a energy wastage.

Our Future work will also explore use of the surrounding resources, present in mobile computing environment, to reduce memory and energy-consumption for our present checkpointing algorithm.

References

1. Elnozahy, E., Johnson, D., Wang, Y.: A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181 (1996)
2. Cao, G., Singhal, M.: Mutable checkpoints: A new checkpointing approach for mobile computing systems. *IEEE Transactions on Parallel and Distributed Systems* **12** (2001)
3. Manivannan, D., Singhal, M.: Failure recovery based on quasi-synchronous checkpointing in mobile computing systems. Technical Report OSU-CISRC-796-TR36, Dept. of Computer and Information Science, The Ohio State University (1996)
4. Myers, D.R.J.A.: Virtual time. *ACM Transactions on Programming Language and Systems* **7** (1985) 404–425
5. Prakash, R., Singhal, M.: Low-cost checkpointing and failure recovery in mobile computing systems. *IEEE Transactions on Parallel and Distributed Systems* **7** (1996) 1035–1048
6. Johnson, D.B., Zawenepoel, W.: Sender-based message logging. In *Proceedings of 17th International Symposium on Fault-Tolerant Computing* (1987) 14–19
7. Johnson, D.B., Zwaenepoel, W.: Recovery in distributed systems using optimistic message logging and checkpointing. (1988) 171–181
8. Manabe, Y.: A distributed consistent global checkpoint algorithm for distributed mobile systems. *8th International Conference on Parallel and Distributed Systems* (2001) 125–132
9. Cao, G., Singhal, M.: Low-cost checkpointing with mutable checkpoints in mobile computing systems. *Proceedings of 18th International Conference on Distributed Computing Systems* (1998) 462–471
10. Singh, P., Cabillic, G.: Fault tolerance and availability in mobile computing environment. selected to appear in *PDPTA* (2003)
11. Manabe, Y.: A distributed first and last consistent global checkpoint algorithm. *12th International Conference on Information Networking* (1998)
12. Singh, P., Cabillic, G.: Successive checkpointing approach for mobile computing environment. selected to appear in *ICWN* (2003)