A NON-FIFO CHECKPOINTING PROTOCOL FOR DISTRIBUTED SYSTEMS

Fuyuan Chao
James R. Kenevan

Department of Computer Science
Illinois Institute of Technology
Chicago, Illinois 60616

Abstract

The checkpointing and rollback recovery are well-known techniques that allow processes to make progress despite of failures. The design of an effective checkpointing protocol in the coordinated approach requires every process to checkpoint its state periodically. A major difficulty is raised due to the fact that processes cannot checkpoint their states at exactly the same instant in a distributed system where there is no common clock or clocks cannot be kept perfectly synchronized. In this paper, we present a checkpointing protocol that does not require the communication subsystem to deliver messages in a first-in first-out fashion.

1. Introduction

Checkpointing and rollback recovery are well-known proper mechanisms for handling failures so that the system can resume the execution after failures with minimal loss of data and time. With this mechanism, a process saves its state on stable storage from time to time by invoking a checkpointing protocol. When a failure occurs, the process rolls back to its most recent saved state and resumes execution by invoking a rollback recovery protocol.

Since checkpointing and rollback recovery are very effective techniques for recovering from failures in a distributed system, it has been widely used and studied by many researchers. However most of the work has been done is useful either for transaction-based systems or FIFO channel systems [1,2,4,5,6,7,9,11,12,13,14]. There is no known checkpointing method designed for non-FIFO systems. Methods that can be used both in general systems and transaction-based systems are of great value. Furthermore, it has been shown [8] that non-FIFO systems are superior to FIFO systems in many ways. Therefore, it is our aim in this paper to provide non-FIFO systems with a proper mechanism so that they can recover from failures.

The rest of this paper is organized as follows: In Section 2, We describe our system model. In Section 3, we present the non-FIFO checkpointing protocol. Section 4 contains the correctness arguments on the protocol. In Section 5, we extend our protocol to deal with failures and concurrent invocations. We do the complexity analysis on our protocol in Section 6. Section 7 contains our conclusion.

2. System Model

The distributed system that we consider has the following features:

(1) It consists of a collection of processes that do not share memory, but communicate by sending or receiving messages through channels.
(2) The channels may not deliver messages in the order sent.
(3) The processes are fail-stop in the sense that they automatically halt in response to any internal failure and do so before the effects of that failure become visible [10].
(4) Any process in the system can fail, but the failure can be detected by all other live processes.
(5) The underlying communication network has great connectivity. The partitioning of the network will not happen by the failures of processes.
(6) Every process has the ability to record the number of messages in its input or output channels. The technique discussed in [8] can be used for this purpose.

The non-FIFO checkpointing protocol that we will present in the following sections is developed under such a distributed system. We do not assume that the underlying system is a transaction-based system. We do not assume
that the processes function deterministically. In particular, we do not assume that an end-to-end transmission protocol is employed to guarantee the FIFO delivery of messages.

3. The Non-FIFO Checkpointing Protocol

The central idea of our checkpointing protocol is based on the observation that a global state is consistent if for every "receive event" included in the global state then the corresponding "send event" has also been included. This observation has been used to define the consistency of a set global states in FIFO channel systems by many researchers [7,8]. Our protocol is similar to the work done by Koo and Toueg [7] with the exception that our protocol does not require the communication subsystem to deliver messages in a FIFO fashion. It has been shown that for any checkpointing protocol to be resilient it must store at least two checkpoints (tentative and permanent) on stable storage [7]. A tentative checkpoint can be undone or changed to be a permanent checkpoint. A permanent checkpoint cannot be undone. Therefore, our checkpointing also takes two kinds of checkpoints during the execution of the protocol. In order to take care of non-FIFO and lossy environments, our checkpointing protocol, like that in [8], assumes that the checkpointing messages are also carried by the successive application messages after the protocol sends its checkpointing messages. However, if a successive application message arrives earlier than the checkpointing message it carries has higher execution priority. Additionally, in order to determine the number of messages in transit between each pair of processes in non-FIFO channels, two extra buffers (input and output) are assigned to each pair of processes for counting messages that were received and sent by processes.

The following terms and notations are used in the description of the protocol.

GCPN: A Global Checkpoint Number is assigned by the initiator process. The initial value of GCPN is set to one and increases by one after the termination of the protocol but before the next invocation. There will be only one GCPN during one invocation of the protocol. The function of GCPN is to ensure that a process takes a tentative checkpoint only once.

LCPN_p: A Local Checkpoint Number is a local variable kept by all other processes p except the initiator process. Initially the value of LCPN_p is set to zero and increases by one after the process has taken a tentative checkpoint or after the process has received a request from its neighboring processes but it does not have to take a tentative checkpoint. After the protocol terminates, the current value of LCPN_p is saved for the use of the next invocation of the protocol. The function of LCPN_p is the same as GCPN.

Willing_to_Checkpoint_p: Every process p keeps this variable to indicate its willingness to take a checkpoint. If for any reason the process p cannot take a checkpoint, the value of willing_to_checkpoint_p is set to "no." Otherwise the value is set to "yes."

I_p: Every process p keeps a list of messages received from all processes q. The initial values of all I_p are set to the current messages in I_p. The main function of I_p is to take care of the messages that may be delivered out of order in non-FIFO channels. It is also used to decide if a process has to take a checkpoint. Moreover, it is used to compute the state of non-FIFO channels (e.g., the message in transition).

O_p: Every process p keeps a list of messages sent to all processes q. The initial values of all O_p are set to the current messages in O_p. The functions of O_p are similar to I_p. Both I_p and O_p can be maintained by the technique discussed in [5].

C_SET_p: means the set of process that p has to send a checkpointing message. Any process r can be included in C_SET_p, if I_p is not empty. The function of C_SET_p is to decide which processes should be sent the checkpointing message.

TRANSIT_p: Every process p maintains this list to record the current channel state from p to all q (for example, the current TRANSIT_p = I_p - O_p). The initial value is set to empty. After the protocol is invoked, the value is calculated by adding the current TRANSIT_p and the previous TRANSIT_p. We consider TRANSIT_p as a part of the checkpoints. It is saved on stable storage and it can be updated only when the checkpoint is updated.

Our non-FIFO checkpointing protocol is patterned on two-phase-commit protocol. It basically works as follows. Let q be a process which initiates the protocol and let p be any process in q's checkpoint set (C_SET_q). In the first phase, process q starts by taking a tentative checkpoint. Then it sends a request "take a tentative checkpoint" with the value of GCPN and the input/output lists (I_q and O_q) to all processes in its checkpoint set. Process q then waits for the replies from all processes p. The reply can be either "yes" or "no," indicating p's acceptance or rejection of q's request. If any reply from p is "no," then
The initiator process q:

Initial State:
GCPN = 1;
I_q = the current content of the input channel in q;
O_q = the current content of the output channel in q;

Begin
Take a Tentative Checkpoint;
For all p in C_SET_q,
   Send(p, "Take a Tentative Checkpoint, I_q, O_q and GCPN");
Reset I_q and O_q;
For all p in C_SET_q,
   Await(p, Willing_to_Checkpoint);
   If there exists a Mi in I_q such that Mi not in TRANSITp:
      Willing_to_Checkpoint_q = "no" or p has failed Then
         Willing_to_Checkpoint_q = "no"
EndIf
If Willing_to_Checkpoint_q = "yes" Then
   Send(t, "Make Tentative Checkpoints Permanent")
Else
   Send(t, "Undo Tentative Checkpoints");
EndIf;
Increment GCPN
End

Figure 1. The Pseudocode for the Initiator Process q

Willing_to_Checkpoint_q is "no." Otherwise Willing_to_Checkpoint_q is "yes." Process q will decide whether to make tentative checkpoints permanent or to undo all tentative checkpoints according to the status of Willing_to_Checkpoint_q. The pseudocode of the initiator process q is shown in Figure 1.

After receiving a request to take a tentative checkpoint from process q, process p will take a tentative checkpoint only if its willingness to take a tentative checkpoint is "yes" (Willing_to_checkpoint_p = yes), the global checkpoint number is greater than p's local checkpoint number (GCPN > LCPN_p) and the condition "If there exists Mi in I_q, such that Mi not in TRANSIT_p, is satisfied. If Willing_to_Checkpoint_q is "yes" and GCPN > LCPN_p, but the condition is not satisfied it increases the value of LCPN_p and immediately sends its willingness to q. Note that the condition is used to force a minimal number of processes to take a tentative checkpoint. The reason that it works can be explained as follows.
of the protocol, the figure in the right-hand side of Figure 2 shows that q takes a new checkpoint at C12, but p does not have to take a new checkpoint since no message has been received by q while its send has not been recorded by p. Figure 3 shows an example in which a process p needs to take a checkpoint. The explanation is similar to that of Figure 2. The figure in the left-hand side of Figure 3 represents the situation before the execution of the protocol. \( I_p \) contains three messages \( M_1, M_2 \) and \( M_3 \); while \( TRANSIT_p \) contains only two messages \( M_1 \) and \( M_2 \). There exists a message \( M_4 \) in \( I_p \) such that \( M_4 \) not in \( TRANSIT_p \). Therefore after q takes a checkpoint at C12, process p has to take a new checkpoint at C12 because there is a message \( M_4 \) received by q while its send has not been recorded by p. The figure in the right-hand side of Figure 3 shows the situation after the execution of the protocol.

On receiving a request to take a checkpoint from the initiator process q, if p takes a tentative checkpoint, it sends a request to all its C_SET_p processes r, and waits for replies from all its C_SET_p processes. If any reply from r is "no," Willing_to_Checkpoint_p becomes "no." Otherwise Willing_to_Checkpoint_p remains unchanged. Then p increases its value of LCPN_p and sends Willing_to_Checkpoint_p to the initiator process q. The pseudocode of the process p is shown in Figure 4.

In the second phase, if the replies from all q’s checkpoint set processes are "yes," the status of Willing_to_Checkpoint_q is "yes." Therefore q decides to make all tentative checkpoints permanent. The decision is sent out by the same fashion as the request is delivered. A process discards its previous permanent checkpoint after it makes its tentative checkpoint permanent or it makes the previous permanent checkpoint a new permanent checkpoint. The pseudocode for this step is shown in Figure 5. Since all or none of the processes take permanent checkpoints, the most recent set of checkpoints is always consistent.

Any other processes p:

Initial State:
- Willing_to_Checkpoint_p = "yes," if p is willing to take a checkpoint. Otherwise Willing_to_Checkpoint_p = "no."
- LCPN_p = 0
- \( I_p \) = the current content of the input channel in p.
- \( O_p \) = the current content of the output channel in p.
- \( TRANSIT_p = \emptyset; \)

On receiving "Take a Tentative Checkpoint, \( I_{qp}, O_{qp} \) and GCPN" from q Do:

Begin
If Willing_to_Checkpoint_p and (GCPN > LCPN_p) Then
If there exists \( M_i \) in \( I_{qp} \) such that \( M_i \) not in \( TRANSIT_p \) Then
Take a Tentative Checkpoint
Compute and save \( TRANSIT_{pq} \);
Reset \( I_p \) and \( O_p \);
For all r in C-SET_p,
Send(r, "Take a Tentative Checkpoint, \( I_{rp}, O_{rp} \) and GCPN"); Reset \( I_{rp} \) and \( O_{rp} \);
For all r in C-SET_p,
Await(r, Willing_to_Checkpoint_r);
If there exists r in C-SET_p,
Willing_to_Checkpoint_r = "no" or r has failed
Then The Willing_to_Checkpoint_p = "no";
Else
Make tentative checkpoints permanent
EndIf
Else
Make tentative checkpoints permanent
EndIf
EndDo

Figure 4. The Pseudocode for All Other Processes p

All processes t:

Initial State:
- Done = false
On first receipt of m = "Make Tentative Checkpoints Permanent" or m = "Undo tentative checkpoints" from q or from its C_SET_q Do:

Begin
If m = "Make Tentative Checkpoints Permanent" and done = false Then
If no tentative checkpoints made Then
Make the previous permanent checkpoint the new permanent checkpoint
Else
Make tentative checkpoints permanent
EndIf
Else
Undo tentative checkpoints
EndIf
Done = true
For all u in C_SET_t, Send(u, m)
EndDo

Figure 5. The Pseudocode for All Processes t
4. Proofs of Correctness

To prove the protocol is correct, we need to show the protocol has the following important properties: 1) The protocol will be terminated. 2) Alter the termination of the protocol, the set of checkpoints saved on stable storage is consistent. 3) The number of processes to take new permanent checkpoints is minimal. We first consider a single invocation of the protocol. We will prove the correctness of the protocol under the assumption that no process fails during the execution of the protocol. Then we will extend the protocol to deal with concurrent invocations and process failures. Let’s begin with the following two definitions.

Definition 1: A process p is said to inherit a checkpointing request from process q, if the following three conditions are satisfied: 1) Willing_to_Checkpointq is "yes," 2) GCPNp > LCPNq, and 3) If there exists M in Iq such that M, not in TRANSITpq.

Definition 2: A set of global states G obtained by the non-FIFO checkpointing protocol is said to be consistent if there exists no M in Iq, such that M, not in Opq, and M, not in TRANSITpq, for any pair of process p and q, where q is the receiving process and p is the sending process.

Lemma 1: Every process inherits at most one request to take a tentative checkpoint during one invocation of the protocol.

Sketch of the proof: Before a process p can inherit a request, its willingness to take a tentative checkpoint must be "yes" and the GCPN must be greater than its LCPNp. After it inherits the request, the value of LCPNp is increased by one (From Figure 1). Therefore, the value of LCPNp for all p in C_SETq, is equal to GCPNp after it takes a tentative checkpoint. Hence, p cannot inherit additional requests to take another tentative checkpoint.

Lemma 2: Every process terminates its execution of the protocol.

Sketch of the proof: The proof is similar to [7]. There are two types of processes executing the non-FIFO checkpointing protocol: 1) some processes execute the protocol without taking a tentative checkpoint, 2) other processes take a tentative checkpoint. Any process that belongs to the first type is clearly terminated. Therefore, the proof is reduced to a proof that for any process belonging to the second type, it does not wait forever for replies from its checkpoint set processes, or for the initiator's decision. Let p be a process of the second type. Let q be a checkpoint set process of p. If q does not inherit a request to take a tentative checkpoint, it immediately sends its willingness to p after receiving p's request. If q inherits p's request, it takes a tentative checkpoint and sends its willingness to p before waiting for the initiator's decision. Therefore, in either case, process p cannot be in a deadlock waiting for replies from q because the straight inherit relation guarantees p inherits a request before q does. Consequently, process p will receive replies from all its C_SET processes. The same argument is also applied to the sending of the initiator's decision message to all processes. Therefore, every process terminates its execution of the protocol.

Theorem 1: Suppose there exists a set of consistent global checkpoints G before the execution of the non-FIFO checkpointing protocol and suppose G' is the new set of checkpoints after the termination of the protocol. Then G' is also a set of consistent global checkpoints.

Sketch of the proof: The proof is by contradiction. Suppose the Theorem is not true, that is G' is not a set of consistent global checkpoints. Then by the definition of a consistent global checkpoint, there must exist M in Iq, such that M, not in Opq, and M, not in TRANSITpq. Then, case 1) either the message M was sent by p before p took its previous checkpoint but arrived in q later and was included in Iq, or case 2) the message M was sent by p after p took its previous checkpoint and arrives normally and is included in Iq. In case 1, the message has to be included in p's TRANSITpq by the construction of the protocol. That is, the message M in TRANSITpq. In case 2, the message has to be included in Opq that is, the message M in Opq. It follows in both cases that there exists no M in Iq, such that M, not in Opq, and M, not in TRANSITpq. This contradicts the assumption that G' is not a set of global consistent checkpoints. Therefore, G' is a set of global consistent checkpoints.

Theorem 2: The number of processes that take new permanent checkpoints during the execution of the protocol is minimal.

Sketch of the proof: Let P0 be the initiator and let S = {P0, 0 < i < k} be the set of processes that take new permanent checkpoints during the execution of the protocol. By Theorem 1, the set of checkpoints taken by processes in S is consistent. Define another set of processes S* = S \ Pk, for any Pk (k > 0) in S. If we could prove that the set of permanent checkpoints taken by processes in S* is still consistent, then S does not contain a minimal number of processes that take permanent checkpoints during the execution of the protocol.

The proof is by contradiction. That is if the set of
permanent checkpoints taken by processes in $S^*$ is not consistent, then $S$ is the minimal number of processes to take permanent checkpoints during the execution of the protocol.

Suppose $S$ is the set of processes that take consistent permanent checkpoints. It can be represented by a tree with the initiator as the root. We may suppose process $p = P_i$ is the process removed from $S$ and process $q = P_j$ in $S$ is either the parent or child process of $p$. In this case the condition "if there exists $M_i$ in $I_{M_i}$ such that $M_i$ not in TRANSIT$_{P_i}$ must have been satisfied in process $p$, but $p$ did not take a checkpoint since it has been removed. Thus there must exist at least one message $M$ that has been received and recorded in $q$ although its send was not recorded by process $p$. By the definition of consistency, the set of checkpoints taken by $S - p (S^*)$ is not consistent. This contradicts the assumption that the set of permanent checkpoints taken by processes in $S^*$ is still consistent. Consequently, $S$ is the minimal number of processes to take a permanent consistent set of checkpoints during the execution of the protocol.

5. Resiliency of the Protocol

Suppose process $p$ is executing the protocol for the initiator $i$ while receiving another checkpointing request from process $q$, but process $q$ is executing the protocol for another initiator $j$. This kind of concurrent invocations can be easily avoided by defining a total ordering of the processes and forcing the processes to initiate the protocol only in the order.

We now consider the possibility that processes fail during the execution of the protocol. There are two situations we must consider. 1) Request failure: a process may fail after sending a request to take a tentative checkpoint. 2) Decision failure: when failures occur, a process may not receive the initiator's decision regarding its tentative checkpoint.

In the case of request failure, since the request was sent before a process failed, the other processes may have received this request and may have taken tentative checkpoints. Therefore when the failed process restarts after failure, its most recent checkpoint on stable storage may be tentative or permanent. The problem is to decide whether to make a tentative checkpoint permanent or to discard it. The decision is made as follows: Suppose that the restarted process is the initiator. The decision is simply to discard all tentative checkpoints. Suppose the process is not the initiator. It can discover the decision by either contacting the initiator or its checkpoint set processes; then it follows the decision to terminate the protocol.

In the case of decision failure, suppose that a process fails before receiving the initiator's decision regarding its tentative checkpoint. When the failed process restarts, if the restarted process discards the tentative checkpoint or makes it permanent, it may violate the initiator's decision. To take care of this situation, our protocol requires the restarted process to wait for the initiator's decision.

6. Complexity Analysis

We first consider the worst case. That occurs when the processes are fully interconnected. We then consider a special structure of processes that can be applied to our protocol to reduce the amount of checkpointing related messages. We do not consider the overhead introduced by the redundant messages carried by the consecutive application messages since it is included as part of the application messages and it is negligible.

The protocol requires two phases to complete the execution. In the first phase, exactly one checkpointing request is sent over each process's output channel. Since the processes are fully interconnected, it requires at most $n(n-1)$ checkpointing request messages and at most $n(n-1)$ reply messages to complete the first phase. In the second phase, exactly one decision message is sent over each process's output channel. It also requires at most $n(n-1)$ decision messages to complete the second phase. Therefore it totally requires $3n(n-1)$ checkpointing related messages to complete the protocol. The complexity of the protocols is $O(n^2)$ in the worst case.

If a spanning tree of processes with the initiator as the root can be determined, the checkpointing messages only have to be sent over the edges of the tree. Since there are $n-1$ edges in the tree, the first phase requires $n-1$ checkpointing messages and $n-1$ reply messages to complete. It also requires $n-1$ decision messages to complete the second phase. Therefore it totally requires $3(n-1)$ checkpointing related messages to complete the protocol. The complexity of the protocols is $O(n)$.

7. Conclusions

We have presented a non-FIFO checkpointing protocol that takes a set of consistent global checkpoints in a non-FIFO communication environment. The non-FIFO checkpointing is non-intrusive. It requires only a minimal number of processes to take a checkpoint, and it only stores two checkpoints on stable storage. In the worst
case, our non-FIFO checkpointing protocol requires $O(n^3)$ checkpointing related messages. But if a special communication structure is designed, the complexity is reduced to $O(n)$. The existence of a set of consistent global states is not sufficient to restart the system properly. Unless it is coordinated with a non-FIFO rollback recovery protocol. The non-FIFO rollback recovery protocol proposed in [3] can be used for this purpose.

8. References


