An optimistic checkpointing and selective message logging approach for consistent global checkpoint collection in distributed systems

Qiangfeng Jiang and D. Manivannan
Department of Computer Science
University of Kentucky
Lexington, KY 40506
{richardj,mani}@cs.uky.edu

Abstract

In this paper, we present an asynchronous consistent global checkpoint collection algorithm which prevents contention for network storage at the file server and hence reduces the checkpointing overhead. The algorithm has two phases: In the first phase, a process initiates consistent global checkpoint collection by saving its state tentatively and asynchronously (called tentative checkpoint) in local memory or remote stable storage if there is no contention for stable storage while saving the state; in the second phase, the message log associated with the tentative checkpoint is stored in stable storage (checkpoint finalization phase). The tentative checkpoint together with the associated message log stored in the stable storage becomes part of a consistent global checkpoint. Under our algorithm, two or more processes can concurrently initiate consistent global checkpoint collection. Every tentative checkpoint will be finalized successfully unless a failure occurs. The finalized checkpoints of each process is assigned a unique sequence number in ascending order. Finalized checkpoints with same sequence number form a consistent global checkpoint.

1 Introduction

Checkpointing and rollback recovery are popular approaches for handling failures in distributed systems. A well designed checkpointing algorithm allows a failed process recover from the recently saved state (called checkpoint) instead of restarting from the very beginning. Existing checkpointing algorithms can be classified into three main categories – *asynchronous*, *synchronous* [2, 5], and *quasi-synchronous* [1, 8]. In *asynchronous* checkpointing, processes take local checkpoints without any coordination. To recover from a failure, a failed process rolls back to its latest checkpoint cp_f and communicates with other processes to find a global checkpoint that is consistent with cp_f. If the current state of some other process P is causally dependent on the checkpoint cp_f, the process P needs to roll back to an earlier checkpoint. When a process other than the failed process needs to roll back to an earlier checkpoint, it may cause further rollbacks of other processes (including the failed process) and this can continue. Therefore, recovery may suffer from *domino effect* in which processes roll back recursively in order to find a consistent global checkpoint. Therefore, *asynchronous* checkpointing is not a storage resource efficient approach. Message logging [4] has been suggested in the literature to cope with the domino effect.

In order to achieve domino-free recovery, *synchronous* checkpointing schemes have been proposed [2, 5]. In this approach, processes synchronize their checkpointing activities by passing explicit control messages so that a globally consistent checkpoint set is always maintained in the system. In many *synchronous* checkpointing algorithms [5], processes may need to block during checkpointing. Therefore, *synchronous* checkpointing has the following disadvantages: Processes have to exchange extra control messages for checkpointing. Some or all processes may have to block their computations for checkpointing, which may degrade the system performance. It may result in several pro-

1-4244-0910-1/07/$20.00 ©2007 IEEE.

cesses taking and storing checkpoints at the stable storage concurrently. Usually, the stable storage is at the network file server and hence it can cause contention for access to stable storage.

However, recovery in *synchronous* checkpointing schemes is simple since processes need only to roll back to the last committed global checkpoint when a failure occurs. Only limited storage space is required for storing the checkpoints. All checkpoints taken before the latest committed global checkpoint can be deleted to save space.

Quasi-synchronous checkpointing (also called *communication-induced* checkpointing) is a hybrid of *asynchronous* and *synchronous* checkpointing schemes. Under *quasi-synchronous* checkpointing algorithms [1, 8], processes are allowed to take local checkpoints independently, and the number of useless checkpoints is minimized by forcing processes to take communication-induced (forced) checkpoints under certain situations. Hence, this class of algorithms overcome the disadvantages of *asynchronous* and *synchronous* checkpointing algorithms, and has the advantages of both types of the algorithms.

Quasi-synchronous checkpointing appears to be an attractive approach for checkpointing in distributed systems. However, existing algorithms in this category have the following drawbacks: Several processes may take checkpoints simultaneously which can cause network contention and hence impact the checkpointing *overhead* and extend the overall execution time [11]. Communication-induced checkpoints have to be taken in general before processing a received message, which may significantly prolong the response time of some received messages. Communication pattern may induce large number of communication-induced checkpoints. Processes have to take their local checkpoints (including communication-induced checkpoints) immediately after specified conditions hold.

Our algorithm reduces/eliminates contention for stable storage by allowing processes to take checkpoints optimistically and store them at stable storage at their own convenience. Moreover, no process needs to take a checkpoint before processing any received message. Each checkpoint taken by our algorithm is composed of a *tentative* checkpoint and a set of messages logged optimistically after taking the *tentative* checkpoint. This mechanism gives processes the liberty of choosing the time to take tentative checkpoints and hence no checkpoint needs to be taken before processing any received message. Furthermore, processes are able to choose their convenient time for writing the tentative checkpoints and the associated message logs to stable storage at the network file server. This helps in minimizing network contention for access to stable storage. Moreover, our algorithm does not incur additional checkpointing overhead due to communication-induced checkpoints unlike the existing algorithms. For example, if each process is required to take checkpoints once in every time interval of t seconds, no process takes more than one checkpoint in any time interval of t seconds.

The rest of the paper is organized as follows. In Section 2 we present the system model and background. Section 3 describes our *quasi-synchronous* checkpointing algorithm. Thereafter, we discuss related work in Section 4 and conclude in Section 5.

2 Background

2.1 System Model

A distributed computation consists of N sequential processes denoted by P_0, P_1, P_2, $\cdots$, and P_{N-1} running concurrently on a set of computers in the network. Processes do not share a global memory or a global physical clock. Message passing is the only way for processes to communicate with one another. The computation is asynchronous: each process evolves at its own speed and messages are transmitted through communication channels, whose transmission delays are finite but arbitrary. Channels need not be FIFO. Messages generated by the underlying distributed computation will be referred to as *application messages*. Explicit control messages generated by the checkpointing algorithm will be referred to as *control messages*. In our algorithm, limited amount of control messages are generated only when necessary.

2.2 Consistent Global Checkpoint

Execution of a process is modeled by three types of events – the send event of a message, the receive event of a message and an internal event. The states of processes depend on one another due to interprocess communication. Lamport's *happened before* relation [6] on events, $\xrightarrow{hb}$, is defined as the transitive closure of the union of two other relations: $\xrightarrow{hb} = (\xrightarrow{xo} \cup \xrightarrow{m})^+$. The $\xrightarrow{xo}$ relation captures the order in which local events of a process are exe-

cuted. The ith event of any process P_p (denoted $e_{p,i}$) always executes before the $(i+1)^{st}$ event: $e_{p,i} \xrightarrow{\text{xo}} e_{p,i+1}$. The $\xrightarrow{\text{m}}$ relation shows the relation between the send and receive events of the same message: if a is the send event of a message and b is the corresponding receive event of the same message, then $a \xrightarrow{\text{m}} b$.

A local checkpoint of a process is a recorded state of the process. A checkpoint of a process is considered as a local event of the process for the purpose of determining the existence of happened before relation among states of processes. Each checkpoint of a process is assigned a unique sequence number. The checkpoint of process P_p with sequence number i is denoted by $C_{p,i}$. We assume that each process takes an initial checkpoint before its execution begins and a final checkpoint after the execution finishes.

The send and the receive events of a message M are denoted respectively by $send(M)$ and $receive(M)$. So, $send(M) \xrightarrow{\text{hb}} C_{p,i}$ if message M was sent by process P_p before taking the checkpoint $C_{p,i}$. Also, $receive(M) \xrightarrow{\text{hb}} C_{p,i}$ if message M was received and processed by P_p before taking the checkpoint $C_{p,i}$. $send(M) \xrightarrow{\text{hb}} receive(M)$ for any message M. The set of events in a process that lie between two consecutive checkpoints is called a checkpointing interval.

A global checkpoint of a distributed computation is a set of checkpoints containing one checkpoint from each process involved in the distributed computation. An orphan message M with respect to a global checkpoint is a message whose $receive(M)$ is recorded in the global checkpoint but the corresponding $send(M)$ is not. A global checkpoint is said to be consistent if there is no orphan message with respect to that global checkpoint. Figure 1 shows two global checkpoints S_1 and S_2. Clearly S_1 is a consistent global checkpoint while S_2 is NOT a consistent global checkpoint since M_5 is an orphan message with respect to the global checkpoint S_2. Next, we present our algorithm.

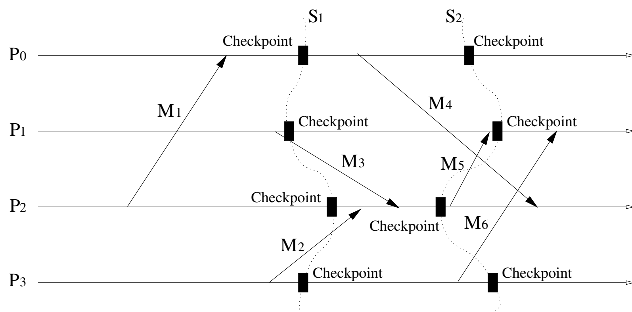

Figure 1. Global checkpoints

3 Algorithm

3.1 Notations

Following are the notations used to describe the algorithm and correctness proof.

$\mathbf{C_{i,k}}$ denotes the (permanent local) checkpoint taken by P_i. It is composed of two parts – a tentative checkpoint $CT_{i,k}$ recording the state of the process and a set of logged messages $logSet_{i,k}$ associated with the checkpoint.

$\mathbf{CT_{i,k}}$ denotes the tentative checkpoint taken by P_i with checkpoint sequence number k. It is usually saved in memory first and then flushed to stable storage after recording the associated log, namely, $logSet_{i,k}$.

$\mathbf{logSet_{i,k}}$ denotes the set of all messages sent and received by P_i after taking the tentative checkpoint $CT_{i,k}$ and before the checkpoint $C_{i,k}$ is finalized. Note that messages are logged optimistically in memory and then flushed to stable storage. Moreover, only messages sent and received after taking a tentative checkpoint and before finalizing the tentative checkpoint. We refer to the operation of flushing the tentative checkpoint and the log of messages to stable storage as *finalizing* the tentative checkpoint. We explain the steps taken for finalizing a tentative checkpoint in Section 3.4.4. Thus, we have $C_{i,k} = CT_{i,k} \cup logSet_{i,k}$.

$\mathbf{CFE_{i,k}}$ denotes the event that represents the finalizing operation of checkpoint $C_{i,k}$. Therefore, all sending and/or receiving events of messages in $logSet_{i,k}$ happen before $CFE_{i,k}$. For any event e of P_i, we have

$$e \xrightarrow{\text{hb}} C_{i,k} \Longleftrightarrow e \xrightarrow{\text{hb}} CFE_{i,k}. \quad (1)$$

$\mathbf{S_k}$ denotes the global checkpoint composed of checkpoints with sequence number k from each process. Thus, $S_k = \{C_{i,k} | i \in \{0, 1, \cdots, N-1\}\}$.

3.2 Basic Idea

The basic idea behind our algorithm is as follows: Any process can initiate taking a consistent global checkpoint. A process accomplishes this by saving its state (called tentative checkpoint) and then piggy-backing this information with each application message it sends after that. When a process P_i receives a message from a process P_j, it comes to know whether P_j has taken a tentative checkpoint as a result of its own consistent global checkpoint initiation or as a result of the initiation of some other process P_k. When

P_i comes to know about the initiation of consistent global checkpoint, it takes a tentative checkpoint if it has not already taken a tentative checkpoint corresponding to this initiation or a concurrent initiation by some other process P_m. Each checkpoint taken is assigned a sequence number which is one more than that assigned to its previous checkpoint. After a process takes a tentative checkpoint, it logs all the messages sent and received in its local memory until it comes to know that all other processes have taken a tentative checkpoint corresponding to its current tentative checkpoint. When a process comes to know that all the processes have taken a tentative checkpoint that corresponds to its current tentative checkpoint, it flushes the associated message log to stable storage; the tentative checkpoint is also flushed to stable storage if it has already done so. Note that the tentative checkpoint can be flushed to stable storage any time after it was taken and before it was finalized. We call the process of flushing a tentative checkpoint and its associated message log into stable storage as "*Finalizing the Checkpoint*". A process is not allowed to initiate a new consistent global checkpoint until it finalizes its current tentative checkpoint. A process, initially, starts in the *normal* status. After a process takes a tentative checkpoint, its status changes from normal to *tentative*. After a tentative checkpoint is finalized, its status changes back to normal. The set of finalized checkpoints with a given sequence number m, denoted by S_m, forms a consistent global checkpoint as proved in Theorem 2. Next, we illustrate the basic idea behind our algorithm with an example.

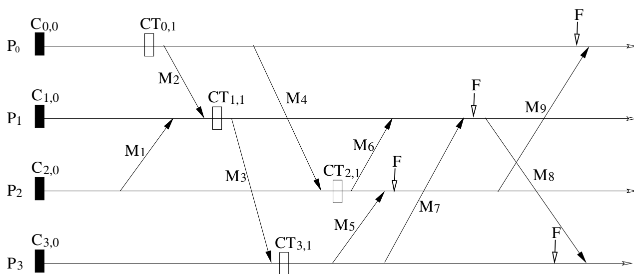

Figure 2. An example illustrating the basic idea behind our algorithm

An Example For explaining the basic idea behind the working of our algorithm, we use the space-time diagram of a distributed computation consisting of four processes shown in Figure 2. P_0, P_1, P_2 and P_3 are the four processes involved in the computation. Initially, their status is *normal* and their initial checkpoints, with sequence number 0, are marked by solid rectangular boxes in the figure. Suppose P_0 initiates consistent global checkpointing by taking a tentative checkpoint $CT_{0,1}$. After taking checkpoint $CT_{0,1}$, it changes its status from *normal* to *tentative* and starts logging all messages sent and received by it until it finalizes this checkpoint. Then, P_0 sends message M_2 to P_1. Upon receiving M_2, P_1 notices that P_0 has taken tentative checkpoint $CT_{0,1}$. Therefore, P_1 takes a tentative checkpoint $CT_{1,1}$ after processing M_2 and P_1's status changes from *normal* to *tentative*. Similarly, P_2 and P_3 take tentative checkpoints $CT_{2,1}$ and $CT_{3,1}$ after receiving messages M_4 and M_3 respectively. P_1 knows that the status of P_0 and P_1 is *tentative* before sending the message M_3; P_1 piggy-backs this information with M_3. Therefore, P_3 knows that the status of P_0, P_1, and P_3 is *tentative* before sending the message M_5. Upon receiving M_5, P_2 knows that the status of all processes is *tentative*. At this point, P_2 finalizes the checkpoint with sequence number 1 by flushing the tentative checkpoint $CT_{2,1}$ and the set of logged messages $\{M_5, M_6\}$ into the stable storage. And we have $C_{2,1} = CT_{2,1} \cup \{M_5, M_6\}$. An "F" mark in the figure indicates the event that finalizes the current tentative checkpoint. After a process finalizes its tentative checkpoint, its status becomes *normal* (after a process takes a tentative checkpoint, it is allowed to take another tentative checkpoint only after finalizing the already taken tentative checkpoint). Similarly, P_1 finalizes its tentative checkpoint after the message M_7 is received. When message M_8 is received, P_3 knows that P_1 has finalized its checkpoint, which indicates that all processes have taken a tentative checkpoint corresponding to its current tentative checkpoint. Therefore, P_3 finalizes its current tentative checkpoint. Note that M_8 should not be included in the set of logged messages in $C_{3,1}$ since it was sent after P_1 finalized $C_{1,1}$. Similarly, P_0 finalizes the checkpoint $C_{0,1}$ upon receiving M_9 without including M_9 in the message log. Now, a consistent global checkpoint $S_1 = \{C_{0,1}, C_{1,1}, C_{2,1}, C_{3,1}\}$ has been recorded.

Some comments In the example given above, there is only one initiator of the consistent global checkpoint S_1. This is primarily to make the example easily understandable. However, under our algorithm, multiple processes can concurrently initiate consistent global checkpointing. A problem with this basic algorithm is that a tentative checkpoint may never be finalized by a process if it does not receive (sufficient) messages from other processes. For ex-

ample, messages such as M_5, M_7, M_8 and M_9 are needed for the four processes to finalize their checkpoints in Figure 2. So, the basic checkpointing algorithm will not work in the absence of sufficient number of application messages that help each process to know the status of every other process in a timely manner. We call this as a consistent global checkpoint *convergence* problem and explain how it can be addressed by using limited number of control messages when necessary in Section 3.5.1. Next, we introduce the data structures needed for presenting the basic algorithm.

3.3 Data Structures

Each process P_i maintains the following data structures.

$\mathbf{csn_i}$: An integer variable containing the sequence number of the current checkpoint of process P_i. The checkpoint representing the initial state of P_i has sequence number 0. P_i sets csn_i to 0 initially. csn_i is increased by one when a new tentative checkpoint is taken.

$\mathbf{stat_i}$: A variable representing the current status of process P_i. The status of a process can be *tentative* or *normal*. The status of a process P_i is updated as follows: P_i's status is set to *normal* initially. P_i's status changes to *tentative* immediately after P_i takes a tentative checkpoint. After P_i knows that the status of all processes is *tentative* (through the information piggy-backed on the application messages), P_i sets its status back to *normal* after finalizing its current tentative checkpoint.

$\mathbf{logSet_i}$: The set of messages logged at P_i after it takes a tentative checkpoint. When $stat_i$ is set to *tentative*, $logSet_i$ is set to empty. Thus $logSet_i$ contains messages sent and received by P_i after a tentative checkpoint is taken. When the status of the process changes from *tentative* to *normal*, the tentative checkpoint and the corresponding $logSet_i$ are flushed to the stable storage.

$\mathbf{tentSet_i}$: The *tentative process set* maintained at P_i. When $stat_i$ is set to *normal*, $tentSet_i$ is set to empty. When P_i takes a tentative checkpoint, P_i sets $tentSet_i$ to $\{P_i\}$. Upon receiving a message, P_i sets $tentSet_i$ to be the union of its current value and the *tentative process set* piggy-backed with the message. This set contains the set of processes that have taken a tentative checkpoint, to the knowledge of P_i.

$\mathbf{allPSet}$: This is the set of all processes, namely, $\{P_0, P_1, \cdots, P_{N-1}\}$.

3.4 The Checkpointing Algorithm

We assume that each process takes an initial checkpoint representing the initial state of the process. The sequence number of the initial checkpoint is set to 0. Moreover, no process is allowed to take a new checkpoint when its status is *tentative*.

3.4.1 Consistent Global Checkpointing Initiation

Any process whose status is *normal* can take a new tentative checkpoint and thereby initiate consistent global checkpointing. When a process P_i takes a tentative checkpoint, it changes its status to *tentative*, increases the checkpoint sequence number csn_i by one and assigns it as the sequence number for the tentative checkpoint, sets $logSet_i$ to empty, and initializes $tentSet_i$ to $\{P_i\}$. After P_i takes a tentative checkpoint, it starts logging all the messages sent and received into $logSet_i$ until its status changes back to *normal.* Csn_i and $tentSet_i$ are piggy-backed with each application message.

3.4.2 Sending Messages

Each process P_i piggy-backs with each application message the current value of csn_i, $stat_i$ and $tentSet_i$. The value of csn_i piggy-backed with messages helps the receiver determine if the sender has initiated a new consistent global checkpointing process. These values piggy-backed with a message M are denoted by $M.csn$, $M.stat$ and $M.tentSet$ respectively. A process receiving a message uses this information piggy-backed with the message to find out about a new checkpoint initiation as well as the processes that have already taken a tentative checkpoint corresponding to this initiation.

3.4.3 Receiving Messages

When process P_i receives a message M from process P_j, it processes the message first and then takes the following actions:

Case (1) $M.stat = stat_i = normal$. In this case, no additional action needs to be taken because neither P_i nor P_j is aware of any new consistent global checkpoint initiation.

Case (2) $M.stat = stat_i = tentative$. Four sub-cases arise:

Sub-case (a) $M.csn < csn_i$. In this case, P_i has already taken and finalized a tentative checkpoint with sequence

number $M.csn$ at the time of receiving M. Therefore, no action needs to be taken.

Sub-case (b) $M.csn = csn_i$. In this case, P_i and P_j have taken checkpoints that belong to the same global checkpoint S_{csn_i}. In order to know how many processes have taken a tentative checkpoint that belongs to the global checkpoint S_{csn_i}, P_i updates $tentSet_i$ to be the union of $tentSet_i$ and $M.tentSet$. If the updated $tentSet_i$ equals to $allPSet$, P_i finalizes its tentative checkpoint since all processes have taken a tentative checkpoint with the same sequence number (i.e., tentative checkpoints that belong to the global checkpoint S_{csn_i}). Section 3.4.4 gives the detailed procedure for finalizing a tentative checkpoint.

Sub-case (c) $M.csn = csn_i + 1$. In this case, P_j finalized the checkpoint with sequence number csn_i before sending the message M and also has taken a tentative checkpoint with sequence number $M.csn$. Therefore, P_i knows that all processes already took a tentative checkpoint that belongs to the global checkpoint S_{csn_i}. Recall that a process is not allowed to take a new tentative checkpoint until it knows all other processes have taken a tentative checkpoint with sequence number equal to that of its current tentative checkpoint and has finalized its current tentative checkpoint. Thus, P_i finalizes its current tentative checkpoint with sequence number csn_i and initiates next consistent global checkpointing by taking a new tentative checkpoint with sequence number $M.csn$.

Sub-case (d) $M.csn > csn_i + 1$. In this case, P_j has finalized the checkpoint with sequence number $csn_i + 1$. Since P_j could have finalized that checkpoint only after all other processes including P_i have taken a tentative checkpoint with sequence number $csn_i + 1$, P_i must have a checkpoint with sequence number greater than or equal to $csn_i + 1$. This is not possible because csn_i is the sequence number of the last tentative checkpoint of P_i. So, this case does not arise.

Case (3) $M.stat = normal$ and $stat_i = tentative$. Three sub-cases arise:

Sub-case (a) $M.csn < csn_i$. In this case, P_i has already taken and finalized a tentative checkpoint with sequence number $M.csn$ at the time of receiving M. Therefore, no further action needs to be taken in this case.

Sub-case (b) $M.csn = csn_i$. In this case, P_j has finalized taking the checkpoint with sequence number csn_i. This means P_j knows that all processes have taken a tentative checkpoint with sequence number csn_i. Hence P_i finalizes its current tentative checkpoint and changes its status back to *normal*.

Sub-case (c) $M.csn > csn_i$. This means P_j has taken a new checkpoint with sequence number $M.csn > csn_i$ and has finalized that checkpoint before P_i finalized the checkpoint with sequence number csn_i. This is impossible because a process cannot finalize a checkpoint with sequence number csn before other processes finalize their checkpoint with sequence number $csn - 1$. So, this case does not arise.

Case (4) $M.stat = tentative$ and $stat_i = normal$. Three sub-cases arise:

Sub-case (a) $M.csn \leq csn_i$. In this case, P_i has already taken and finalized a tentative checkpoint with sequence number $M.csn$ at the time of receiving M. So, the message is simply processed without taking any additional action.

Sub-case (b) $M.csn = csn_i + 1$. In this case, P_j has taken a new tentative checkpoint about which P_i comes to know through M for the first time. Therefore, P_i takes a tentative checkpoint with sequence number $M.csn$. The procedure for taking a new tentative checkpoint is same as that in Section 3.4.1. In addition to that, P_i updates $tentSet_i$ to be the union of $tentSet_i (= \{P_i\})$ and $M.tentSet$ in the message. Thus, P_i gets P_j's knowledge about the processes that have taken a tentative checkpoint with sequence number $csn_i + 1$.

Sub-case (c) $M.csn > csn_i + 1$. This is similar to the **sub-case (d)** under **case (2)**.

3.4.4 Finalizing a Tentative Checkpoint that belongs to a Consistent Global Checkpoint with a Given Sequence Number

If the status of a process P_i is *tentative* and it knows (through the messages received from other processes) that the status of all processes in the system is tentative (i.e. $tentSet_i = allPSet$), it flushes its current tentative checkpoint (the most recent tentative checkpoint taken) and also the associated message log $logSet_i$, into the stable storage and makes it permanent. The tentative checkpoint together with the message log stored is called a checkpoint of the process and it is assigned the same sequence number as the tentative checkpoint stored. This checkpoint together with the checkpoints with same sequence number from all other processes forms a consistent global checkpoint, as proved in Theorem 2.

Formal description of the basic checkpointing algorithm is given in Figure 3.

When P_i starts
$csn_i = 0$; $stat_i = normal$; */* Initialization */*

Procedure: takeTentativeCheckpoint(i: integer)
$csn_i = csn_i + 1$; $stat_i = tentative$;
$tentSet_i = \{P_i\}$; */* Include the process id in the set */*
$logSet_i = \emptyset$; */* Initialize the message log to empty set */*
Take tentative checkpoint CT_{i,csn_i};

When P_i starts to take a checkpoint
takeTentativeCheckpoint(i);

When P_i sends a message M to P_j
$M.csn = csn_i$; */* Piggy-back current info with the message */*
$M.stat = stat_i$;
$M.tentSet = tentSet_i$;
if $stat_i == tentative$ **then** $logSet_i = logSet_i \cup \{M\}$;

When P_i receives a message M from P_j
Process M;
if $stat_i == normal$ **then**
  if $M.stat == tentative$ **then**
    if $M.csn == csn_i + 1$ **then** */* a new consistent global ckpt */*
      takeTentativeCheckpoint(i);
      $tentSet_i = M.tentSet \cup tentSet_i$;
else if $stat_i == tentative$ **then**
  $logSet_i = logSet_i \cup \{M\}$; */* Log the received message */*
  if $M.stat == normal$ **then**
    if $M.csn == csn_i$ **then** */* P_j has finalized the ckpt C_{j,csn_i} */*
      $stat_i = normal$;
      Flush $logSet_i - \{M\}$ and CT_{i,csn_i} to the stable storage;
  else if $M.stat == tentative$ **then**
    if $M.csn == csn_i$ **then** */* took ckpt before sending the msg */*
      $tentSet_i = M.tentSet \cup tentSet_i$;
      if $tentSet_i == allPSet$ **then** */* Each proc took the ckpt */*
        $stat_i = normal$;
        Flush $logSet_i$ and CT_{i,csn_i} to the stable storage;
    else if $M.csn == csn_i + 1$ **then** */* took a new tentative ckpt */*
      $stat_i = normal$;
      Flush $logSet_i - \{M\}$ and CT_{i,csn_i} to the stable storage;
      takeTentativeCheckpoint(i);
      $tentSet_i = M.tentSet \cup tentSet_i$;

Figure 3. The Basic Checkpointing Algorithm

3.5 Optimizations

3.5.1 A Convergence Problem

As we noted earlier, the basic checkpointing algorithm presented in the previous section may not converge if enough messages are not exchanged. To address this problem, we present a mechanism that utilizes control messages to expedite convergence when necessary. So, control messages are used only if a tentative checkpoint has not been finalized within a predetermined period of time. In the following, we discuss a mechanism to introduce limited amount of control messages to expedite convergence when necessary. We introduce three types of control messages – checkpoint begin *CK_BGN* message, checkpoint request (*CK_REQ*) and checkpoint end (*CK_END*) messages. A process P_i sets a timer when it takes a tentative checkpoint. If P_i does not finalize its tentative checkpoint before the timer expires, it sends a *CK_BGN* message to a pre-specified process, e.g. P_0. Upon receiving the message, P_0 takes a tentative checkpoint if it has not taken yet taken and then sends a *CK_REQ* message to P_1, P_1 does the same and sends it to P_2, etc. and finally *CK_REQ* reaches back to P_0. After P_0 receives the message back, it sends *CK_END* message to all the processes. When a process receives the *CK_END* message, it finalizes its local checkpoint with the sequence number contained in the *CK_END* message if it has not already finalized it; it ignores the message if it has already finalized. Control messages are not sent if each global checkpoint can be finalized within the timeout interval. The *tentative process set* can be used to further reduce the number of control messages as follows:

Case (1) Limiting the number of *CK_BGN* messages. As we know, one *CK_BGN* message is enough to notify P_0 to initiate *CK_REQ* messages for each global checkpoint. In the method described above every process that times out sends *CK_BGN* to P_0. Such redundant messages can be reduced using the information contained in *tentative process set*. Suppose it is time for P_i to send a *CK_BGN* message to P_0. Before sending the message, it checks if there is a process P_j that belongs to $tentSet_i$ such that $j < i$. If such a P_j exists, P_i does nothing since it knows that P_j or some other process with process id smaller than j will send a *CK_BGN* message to P_0. Otherwise, P_i sends a *CK_BGN* message to P_0. Clearly, this method reduces the number of *CK_BGN* messages. However, it introduces a new problem, namely, the process with lower id may have finalized the checkpoint already and has not exchanged any message afterwards. This way, P_i may not be able to finalize the checkpoint. This problem can be solved by requiring P_0 always broadcast a *CK_END* message to all other processes when it finalizes a checkpoint.

Case (2) Saving *CK_REQ* messages. Under the simple version of the approach of forwarding the *CK_REQ* message, every process needs to forward it once. However, the number of *CK_REQ* messages can be further reduced by the following method. Suppose it is time for P_i to forward the message. If it has finalized this checkpoint, it forwards the message to P_0 directly. Otherwise, P_i looks for a process P_j for which the following condition holds.

$$(j > i)\ AND\ (P_j \notin tentSet_i)\ AND\ (\forall k \in \{z | i < z < j\}, P_k \in tentSet_i)$$

If such a process P_j is found, P_i forwards the message to P_j because all processes with process numbers greater than i and less than j have already taken a tentative checkpoint and there is no need to ask them to take it again. Otherwise, all processes with process numbers greater than i have already taken a tentative checkpoint. Therefore, P_i forwards the message to P_0 directly.

Figure 4 gives the formal description of how control messages can be used to augment the basic algorithm to help convergence. In this we use CM to denote a control message. A CM has two fields, namely, *type* and *csn*. *CM.type* can have one of the three values, namely, *CK_BGN*, *CK_REQ* or *CK_END*. *CM.csn* is the sequence number of the current tentative checkpoint of the sender when it sends the control message CM. $CM(atype, acsn)$ refers to the control message CM with $CM.type = atype$ and $CM.csn = acsn$. For example, $CM(CK_BGN, 3)$ refers to a control message *CK_BGN* with $csn = 3$ piggybacked with it.

A timer is used by each process to determine when to send control messages as follows: A process sets a timer when it takes a tentative checkpoint. When the timer expires, it initiates sending a control message CM. The timer is canceled when a process finalizes the checkpoint or it receives a CM with sequence number equal to that of its current tentative checkpoint.

We illustrate how control messages help in convergence with the help on the example shown in Figure 5. Suppose P_1 takes a tentative checkpoint $CT_{1,1}$ first and sends a message M_2 to P_2. Upon receiving M_2, P_2 takes a tentative checkpoint $CT_{2,1}$. When the timer set for $CT_{1,1}$ expires, P_1 sends a *CK_BGN* message (CK_BGN_1) to P_0 (P_2 does not send a *CK_BGN* message since it knows that P_1 will send such message to P_0). Upon receiving CK_BGN_1, P_0 takes a tentative checkpoint $CT_{0,1}$ and sends a *CK_REQ* message CK_REQ_1 to P_1. Thereafter, P_1 sends a *CK_REQ* message CK_REQ_2 to P_3 since it knows that P_2 has already taken $CT_{2,1}$. Finally, the *CK_REQ* message CK_REQ_3 returns to P_0. Now, P_0 knows that all processes have already taken a tentative checkpoint with sequence number 1. Therefore, it finalizes its current tentative checkpoint and broadcasts a *CK_END* message to every other process and flushes logged application messages and $CT_{0,1}$ to the stable storage. Upon receiving CK_END, P_1, P_2 and P_3 flush their logged messages and tentative checkpoints with sequence number 1 respectively. This way, all processes finalize the checkpoints with sequence number 1 and return to *normal* status in finite time. Without these control messages, the original algorithm does not converge in this example. Although P_3 sends out messages such as M_5 and M_6, it does not receive any message. Therefore, P_3 is unable to obtain the status information of other processes, and hence P_3 can not finalize its tentative checkpoint $CT_{3,1}$.

When the timer for finalizing the tentative checkpoint on P_i expires
  if $i == 0$ **then** /* P_0 *initiates* CK_REQ *messages directly* */
    forwardCheckpointRequest(P_0, CM);
  else /* $i = 1, 3, \cdots, or N-1$ */
    for each $P_k \in tentSet_i$ **do**
      if $k < i$ **then** return;
    Send $CM(CK_BGN, csn_i)$ to P_0;
Procedure: forwardCheckpointRequest(P_i, CM)
  if $i == N-1$ **then** k = 0; /* P_{N-1} *forwards* CK_REQ *message to* P_0 */
  else /* *looks for* P_j *such that status of* $P_{i+1}, \cdots, and\ P_{j-1}$ *is tentative* */
    for $k = i + 1$ **to** $N - 1$ **do**
      if $P_k \notin tentSet_i$ **then** break;
    if $P_k \in tentSet_i$ **then** $k = 0$;
  Send $CM(CK_REQ, csn_i)$ to P_k;

When P_i receives CM from P_j
  if $CM.csn == csn_i + 1$ **then**
    if $stat_i == tentative$ **then**
      Flush $logSet_i$ and CT_{i,csn_i} to the stable storage;
    takeTentativeCheckpoint(i);
    forwardCheckpointRequest(P_i, CM);
  else if $CM.csn == csn_i$ **then**
    if $CM.type == CK_BGN$ **then**
      if $stat_i == tentative$ **then**
        if $CM(CK_REQ, csn_i)$ has been sent **then** return;
        forwardCheckpointRequest(P_i, CM);
      else if $CM(CK_END, csn_i)$ has not been sent **then**
        Send $CM(CK_END, csn_i)$ to $P_1, P_2, \cdots$, and P_{N-1};
    else if $CM.type == CK_REQ$ **then**
      if $i == 0$ **then** /* P_0 *initiates* CK_END *if necessary* */
        if $CM(CK_END, csn_i)$ has been sent **then** return;
        Send $CM(CK_END, csn_i)$ to $P_1, P_2, \cdots$, and P_{N-1};
        if $stat_i == tentative$ **then**
          $stat_i = normal$;
          Flush $logSet_i$ and CT_{i,csn_i} to the stable storage;
      else forwardCheckpointRequest(P_i, CM);
    else if $stat_i == tentative$ **then** /* $CM.type == CK_END$ */
      $stat_i = normal$;
      Flush $logSet_i$ and CT_{i,csn_i} to the stable storage;

Figure 4. Augmenting the Basic Algorithm with Control Messages to Speed up Convergence

3.6 Correctness Proof

We refer to the checkpointing algorithm with control messages as the *generalized checkpointing algorithm*. With this definition, we have Theorem 1.

Theorem 1 *The generalized checkpointing algorithm con-*

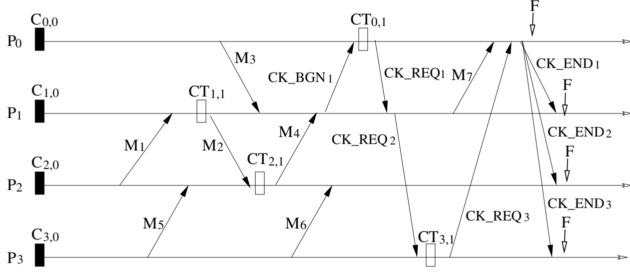

Figure 5. An example illustrating the use of control messages in the algorithm

verges, i.e. after a process takes a tentative checkpoint with a given sequence number csn, every process eventually finalizes a checkpoint with sequence number csn.

Proof. We prove this by contradiction. Suppose the generalized checkpointing algorithm does not converge. In other words, there is at least one process, say P_i, that takes a tentative checkpoint $CT_{i,k}$ but never finalizes the checkpoint $C_{i,k}$. Depending upon why P_i takes $CT_{i,k}$, the following two cases arise.

Case (1) P_i takes $CT_{i,k}$ because it receives a message $CM(CK_REQ, k)$ from a process P_i. Upon receiving such a message, P_i needs to forward the message to a process P_h and assure that all processes with process number greater than i and less than h have already taken a tentative checkpoint with sequence number k. This is repeated until the message returns to P_0 (P_{N-1} forwards the message to P_0 or some process P_j ($j < N-1$) forwards it to P_0 directly since P_j knows that all processes with process number greater than j have taken a tentative checkpoint with sequence number k). Once P_0 receives the message, it finalizes $C_{0,k}$ and broadcasts a message $CM(CK_END, k)$ to all other processes. Upon receiving this message, each process finalizes its tentative checkpoint with sequence number k if appropriate. In particular, P_i finalizes $C_{i,k}$ which is a contradiction to our assumption.

Case (2) P_i takes $CT_{i,k}$ due to other reasons. Then a timer is set when $CT_{i,k}$ is taken at P_i. If the timer is canceled due to receiving a *CK_REQ* or *CK_END* message with sequence number k, P_0 has initiated a message $CM(CK_REQ, k)$. Otherwise, P_i or some process with process number smaller than i will send a message $CM(CK_BGN, k)$ to P_0. Therefore, P_0 will receive at least one *CK_BGN* message with sequence number k. Then P_0 initiates the process of forwarding *CK_REQ* messages. Similar to **Case(1)**, P_i finalizes the checkpoint $C_{i,k}$ which is a contradiction to our assumption.

Hence the theorem. □

Theorem 2 *For each k, the set $S_k = \{C_{i,k} | i \in 0, 1, \cdots, N-1\}$ forms a consistent global checkpoint.*

Proof. We prove this by contradiction. Suppose S_k is not consistent. Then, there exists a message M, sent from P_i to P_j (for some $i, j \in \{0, 1, \cdots, N-1\}, i \neq j$), such that $C_{i,k} \xrightarrow{hb} send(M)\ AND\ receive(M) \xrightarrow{hb} C_{j,k}$.

Depending on the receiving time of the message M, the following two cases arise.

Case (1) $receive(M) \xrightarrow{hb} CT_{j,k}$ (a). Since $C_{i,k} \xrightarrow{hb} send(M)$, $CFE_{i,k} \xrightarrow{hb} send(M)$ (b). Since P_i has finalized $C_{i,k}$ before sending M, P_i already knew that all processes have taken tentative checkpoints with sequence number k. In particular, before finalizing $C_{i,k}$, P_i knew that P_j took $CT_{j,k}$. Hence, $CT_{j,k} \xrightarrow{hb} CFE_{i,k}$ (c). From (a), (b) and (c), we have $receive(M) \xrightarrow{hb} CT_{j,k} \xrightarrow{hb} CFE_{i,k} \xrightarrow{hb} send(M)$, i.e. $receive(M) \xrightarrow{hb} send(M)$, a contradiction.

Case (2) $CT_{j,k} \xrightarrow{hb} receive(M) \xrightarrow{hb} CFE_{j,k}$ (a). Similar to **Case (1)**, we have $CFE_{i,k} \xrightarrow{hb} send(M)$. Upon receiving M, P_j knows that P_i has finalized the checkpoint $C_{i,k}$. Therefore, it knows that all other processes have taken a tentative checkpoint with sequence number k. Based on this information, P_j finalizes the checkpoint $C_{j,k}$ not including message M in the checkpoint. Therefore, we have $CFE_{j,k} \xrightarrow{hb} receive(M)$ (b). From (a) and (b) we have $receive(M) \xrightarrow{hb} receive(M)$ which is a contradiction.

Hence the theorem. □

4 Related Work

In this section, we briefly review previously proposed algorithms related to our checkpointing algorithm.

In order to minimize the network contention caused by storing local checkpoints to the stable storage simultaneously, Plank and Vaidya proposed two *synchronous* checkpointing algorithms. These algorithms are referred to as staggered checkpointing algorithms. Plank's [10] algorithm is a variation of the Chandy-Lamport algorithm [3] that staggers a *limited* number of checkpoints depending on the network topology. However, a completely connected topology would subvert staggering in this algorithm. Based on Plank's algorithm, Vaidya proposed a *synchronous* checkpointing algorithm that staggers all checkpoints. Similar to the algorithms of Chandy *et al.* [3] and Plank [10],

Vaidya's algorithm uses a coordinator to initiate the consistent global checkpoint collection. Under Plank's algorithm, the *take_checkpoint* message sent by the coordinator propagates to all processes involved in the distributed computation one by one and returns back to the coordinator finally. Upon receiving the *take_checkpoint* message, a process takes a physical checkpoint if it has already not taken one. After the physical checkpoint is taken, the message is forwarded to the next process. This way, no two processes will take checkpoints simultaneously and hence prevents contention for stable storage. Manivannan *et al.* proposed a *quasi-synchronous* checkpointing algorithm [7] which staggers checkpoints to prevent two or more processes from taking a checkpoint at the same time. Zhang *et al*. [12] and Oliner *et al*. [9] discussed checkpointing performance issues in large-scale cluster systems.

5 Conclusion

In this paper, we presented a novel *quasi-synchronous* checkpointing algorithm that makes every checkpoint belong to a consistent global checkpoint. Under this algorithm, every process takes tentative checkpoints and optimistically logs messages received after a tentative checkpoint is taken and before the tentative checkpoint is finalized. Since a tentative checkpoint can be taken any time in the contention for stable network storage that arises due to several processes storing the checkpoints simultaneously is minimized by allowing processes to store the tentative checkpoints in local memory; tentative checkpoints taken can be flushed to stable storage anytime before that checkpoint is finalized. Moreover, unlike existing communication-induced checkpointing algorithms, our algorithm, generally, does not force a process to take a checkpoint before processing any received message in order to prevent useless checkpoints. Thus, a process can first process the received message and then take checkpoint. This improves the response time for messages. It also helps a process take the regularly scheduled basic checkpoints at the scheduled times. We did not include the results of our performance evaluation due to space limitation.

Acknowledgment

The authors thank the reviewers for their valuable comments which helped in improving the content and presentation of the paper. This material is based in part upon work supported by the US National science Foundation under Grant No. IIS-0414791 and the US Department of Treasury Award #T0505060. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Department of Treasury.

References

[1] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. A Communication Induced Algorithm that Ensures the Rollback Dependency Trackability. In *Proceedings of the* 27th *International Symposium on Fault-Tolerant Computing*, July 1997.

[2] G. Cao and M. Singhal. Checkpointing with mutable checkpoints. *Theoretical Computer Science*, 290(2):1127–1148, January 2003.

[3] K. M. Chandy and L. Lamport. Distributed Snapshots : Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1), 1985.

[4] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[5] R. Koo and S. Toueg. Checkpointing and Roll-back Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[6] L. Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM.*, 21(7), 1978.

[7] D. Manivannan, Q. Jiang, J. Yang, K. Persson, and M. Singhal. "An Asynchronous Recovery Algorithm based on a Staggered Quasi-Synchronous Checkpointing Algorithm". In *Lecture Notes in Computer Science Series No.3741*, pages 117–128, Springer Verlag, December 2005.

[8] D. Manivannan and M. Singhal. Asynchronous Recovery Without Using Vector Timestamps. *Journal of Parallel and Distributed Computing*, 62(12):1695–1728, December 2002.

[9] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta. Performance implications of periodic checkpointing on large-scale cluster systems. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18*, page 299.2, Washington, DC, USA, 2005. IEEE Computer Society.

[10] J. S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Priceton University, June 1993.

[11] N. Vaidya. Staggered Consistent Checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):694–702, January 1999.

[12] Y. Zhang, M. S. Squillante, A. Sivasubramaniam, and R. K. Sahoo. Performance Implications of Failures in Large-Scale Cluster Scheduling. In *JSSPP '04: Job Scheduling Strategies for Parallel Processing, 10th International Workshop*, pages 233–252, 2004.