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

Qiangfeng Jiang, Yi Luo, D. Manivannan *
Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA

ARTICLE INFO
Keywords:
Distributed systems
Fault-tolerance
Checkpointing
Rollback recovery
Communication-induced checkpointing

ABSTRACT
Checkpointing and rollback recovery are widely used techniques for achieving fault-tolerance in distributed systems. In this paper, we present a novel checkpointing algorithm which has the following desirable features: A process can independently initiate consistent global checkpointing by saving its current state, called a tentative checkpoint. Other processes come to know about a consistent global checkpoint initiation through information piggy-backed with the application messages or limited control messages if necessary. When a process comes to know about a new consistent global checkpoint initiation, it takes a tentative checkpoint after processing the message (not before processing the message as in existing communication-induced checkpointing algorithms). After a process takes a tentative checkpoint, it starts logging the messages sent and received in memory. When a process comes to know that every other process has taken a tentative checkpoint corresponding to current consistent global checkpoint initiation, it flushes the tentative checkpoint and the message log to the stable storage. The tentative checkpoints together with the message logs stored in the stable storage form a consistent global checkpoint. Two or more processes can concurrently initiate consistent global checkpointing by taking a new tentative checkpoint; in that case, the tentative checkpoints taken by all these processes will be part of the same consistent global checkpoint. The sequence numbers assigned to checkpoints by a process increase monotonically. Checkpoints with the same sequence number form a consistent global checkpoint. We also present the performance evaluation of our algorithm.

1. Introduction

Checkpointing and rollback recovery are popular approaches for handling failures in distributed systems. A well-designed checkpointing algorithm allows a failed process to 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 — uncoordinated, coordinated [4,15,16,18,25], and communication-induced [2,14,20,21]. In uncoordinated checkpointing, processes take local checkpoints without any coordination. To recover from a failure, the failed process determines a consistent global checkpoint by communicating with other processes and all the processes rollback to that consistent global checkpoint. Message logging [12,13,26,27] has been suggested in the literature to cope with the domino effect. Since multiple checkpoints are stored, uncoordinated checkpointing is not a storage resource efficient approach. In order to achieve domino-free recovery, coordinated checkpointing schemes have been proposed [4,15,16,18,25]. In this approach, processes synchronize their checkpointing activities by passing explicit control messages so that a globally consistent checkpoint is always maintained in the system. Communication-induced checkpointing is a hybrid of uncoordinated and coordinated checkpointing schemes. Under communication-induced checkpointing algorithms [2,14,20–22], 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 uncoordinated and coordinated checkpointing algorithms, and have the advantages of both coordinated and uncoordinated checkpointing algorithms.

Communication-induced 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 [28,29]. In general, communication-induced checkpoints have to be taken before processing a received message, which may significantly prolong the response time of those corresponding received mes-
sages. Moreover, 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.

We use the term “Optimistic” [1,7] because our algorithm saves checkpoints and message logs in memory first, and then flushes them to stable storage to prevent contention for stable storage. Each checkpoint taken by our algorithm is composed of a tentative checkpoint representing the state of the process and a set of messages logged 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 overhead due to communication-induced checkpoints, unlike the existing algorithms.

The rest of the paper is organized as follows. First, we discuss related work in Section 2. In Section 3 we present the system model and background. Then, Section 4 describes our communication-induced checkpointing algorithm and the recovery algorithm. We present the performance evaluation of our checkpointing algorithm and also compare our algorithm with one other algorithm in Section 5. Thereafter, we conclude in Section 6.

2. Related work

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

Barigazzi and Strigini [3] presented a coordinated checkpointing algorithm in which they assume that all communications between processes are atomic. Koo and Toueg [16] relaxed this assumption. Some other works [15,16] have focused on minimizing the number of synchronization messages and the number of forced checkpoints during checkpointing. These algorithms force relevant processes in the system to block their computations during the checkpointing process, which will degrade system performance [8]. Chandy and Lamport [5] presented the first non-blocking algorithm for coordinated checkpointing. However, it leads to a message complexity of $O(n^2)$. Silvas [25] also addressed this issue and presented another non-blocking algorithm.

Cao and Singhal [4] presented a min-process and non-blocking algorithm. The non-blocking algorithm is based on the concept of “mutable checkpoint”, which can be saved anywhere, e.g. the main memory or the local disk. Therefore, the algorithm avoids the overhead of transferring “mutable checkpoints” to the stable storage at the file server across the network. Moreover, it forces only a minimum number of processes to save their checkpoints on the stable storage. Mandal and Mukhopadhyaya [19] presented a checkpointing algorithm in which processes are arranged in a ring. Processes are allowed to take checkpoints independently anytime in a predetermined time interval, called total checkpointing time (TCT). Once a process takes a checkpoint, it sends a checkpoint request to the next process in the ring. A process receiving the checkpoint request has to take a checkpoint if it did not take a checkpoint in that interval so far and then forwards the checkpoint request to the next process in the ring and this continues. There are two drawbacks with this algorithm. One is that clocks need to be synchronized so that each process has the same view of the checkpoint interval. The other problem is, if a process takes a checkpoint early in the interval TCT, it will force all other processes to take checkpoints sequentially which will cause contention at stable storage. In our algorithm, a process does not send any control message for taking checkpoints unless it is necessary. Moreover, when a process receives a message from a process that already took a tentative checkpoint, it does not have to take a checkpoint immediately; it can take checkpoint after processing the message. In addition, the checkpoint taken need not be flushed immediately to stable storage, thus preventing contention for stable storage.

The problem of network contention caused by storing local checkpoints to the stable storage simultaneously can significantly impact the checkpointing overhead and extend the total execution time of the distributed computation [28,29]. Contention for stable storage can be mitigated by staggering the checkpoints [24]. Staggered checkpointing attempts to prevent two or more processes take checkpoints at the same time, thereby reducing contention for stable storage. To the best of our knowledge, checkpoint staggering has previously been proposed only for synchronous, or coordinated, checkpointing algorithms [24,29]. These algorithms are referred to as staggered checkpointing algorithms. Plank [24] proposed a variation of the Chandy-Lamport algorithm [5] that staggered 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 observation, Vaidya [29] proposed another coordinated checkpointing algorithm that staggered all checkpoints. Like Plank [24] and Chandy-Lamport [5], Vaidya's algorithm [29] uses a coordinator to initiate the checkpointing process. It has two phases. In the first phase, the coordinator $P_0$ takes a physical checkpoint (i.e., saves its current state in stable storage) and sends a take_checkpoint message to the next process $P_1$. Upon receipt of the take_checkpoint message, process $P_1$ takes a physical checkpoint and resends it to process $P_i$, where $i > 0$ and $j = (i+1) \mod n$. The phase is terminated when the coordinator $P_0$ receives the take_checkpoint message from the last process $P_{n-1}$. In the second phase, the channel states, called logical checkpoints, are recorded. The set of logical checkpoints, together with the physical checkpoints, form a consistent global state. The algorithm successfully staggered all physical checkpoints. However, as shown in our simulation results, contention for stable storage always exists for taking the logical checkpoints. In terms of the number of collisions due to the logged messages, Vaidya's algorithm [29] always performs worse, compared to our algorithm.

3. Background

3.1. System model

A distributed computation consists of $N$ sequential processes denoted by $P_0$, $P_1$, $P_2$, $\ldots$, 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 are assumed to be FIFO and the computation is assumed to be piecewise-deterministic [7,9]. Elnozahy et al. [7] present an excellent survey of the state of the art in checkpointing and recovery. Messages generated by the underlying distributed computation will be referred to as application messages. Explicit control messages generated by checkpointing algorithm will be referred to as control messages. In our algorithm, limited amount of control messages are generated for the collection of consistent global checkpoint, only when necessary.

3.2. Consistent global checkpoint

The 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 re-
lation [17] 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
executed. The \(i\)th event of any process \(P_i\) (denoted \(e_{p,i}\)) always ex-
ecutes before the \((i + 1)st\) event: \(e_{p,i} \xrightarrow{m} e_{p,i+1}\). The \(\xrightarrow{m}\)
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 corre-
spanding receive event of the same message, then \(a \xrightarrow{m} b\) [21,23].

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 checkpoints of processes. Each checkpoint
of a process is assigned a unique sequence number. The checkpoint
of process \(P_i\) with sequence number \(i\) is denoted by \(C_{p,i}\).

The send and the receive events of a message \(M\) are denoted
respectively by send\((M)\) and receive\((M)\). So, send\((M) \xrightarrow{hb} C_{p,i}\) if
message \(M\) was sent by process \(P_i\) before taking the checkpoint \(C_{p,i}\).
Also, receive\((M) \xrightarrow{hb} C_{p,i}\) if message \(M\) was received and processed
by \(P_j\) before taking the checkpoint \(C_{p,j}\). send\((M) \xrightarrow{hb} receive(M)\)
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. Fig. 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 \(S_2\)
is an orphan message with respect to \(S_2\).

Next, we present our algorithm.

4. Algorithm

4.1. Notations

Following are the notations used in describing the algorithm and
its correctness proof.

- \(C_{i,k}\) 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.
  - \(CT_{i,k}\) 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, \(logSet_{i,k}\) or whenever there is no
    contention for accessing stable storage.
  - \(logSet_{i,k}\) denotes the set containing 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. 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 4.4.4.

Thus, we have \(C_{i,k} = CT_{i,k} \cup logSet_{i,k}\).

- \(CEF_{i,k}\) 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 \(CEF_{i,k}\). For any event \(e\)
of \(P_i\), we have

\[
e \xrightarrow{hb} C_{i,k} \iff e \xrightarrow{hb} CEF_{i,k}.
\]  

4.2. 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 a 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.
When \(P_i\) comes to know about a new initiation of consistent
global checkpoint, it takes a tentative checkpoint. Each checkpoint
taken is assigned a sequence number which is one more than its
previous checkpoint. After a process takes a tentative checkpoint,
it continues logging 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 its current tentative checkpoint
(if it has not already done so) and the associated message log to stable
storage. 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.

4.2.1. An example

For explaining the basic idea behind our algorithm, we use
the space-time diagram of a distributed computation consisting of
four processes shown in Fig. 2. \(P_0, P_1, P_2\) and \(P_3\) are the four
processes involved in the computation. Initially, the status of each
process 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 \(CT_{0,0}\). After taking checkpoint \(CT_{0,0}\), it changes
its status from normal to tentative and starts logging in memory all
messages sent and received by it until it finalizes this checkpoint.
Then, \(P_0\) sends a message \(M_2\) to \(P_1\). Upon receiving \(M_2, P_1\) notices
that \(P_0\) has taken \(CT_{0,0}\). Therefore, \(P_1\) takes a tentative checkpoint
\(CT_{1,1}\) after processing \(M_2\) and \(P_i\)'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 \(M_5\) 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 \(P_0, P_1, P_2\) and \(P_3\) is tentative before
sending the message \(M_5\). Upon receiving \(M_5, P_3\) knows that the
status of all processes is tentative. At this point, \(P_i\) finalizes the
checkpoint with sequence number 1 by flushing the tentative
checkpoint \(CT_{2,1}\) (if it has not already done so) and the set of
logged messages \(\{M_3, M_5\}\) into the stable storage. And we have
\(C_{2,1} = CT_{2,1} \cup \{M_3, M_5\}\). An “F” mark in the figure indicates
the event of finalizing 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

\[
S_k = \{C_{i,k} | i \in \{0, 1, \ldots, N - 1\}\}
\]
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 $M_6$ 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 by taking a tentative checkpoint. 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 example, messages such as $M_5$, $M_7$, $M_8$ and $M_9$ are needed for the four processes to finalize their checkpoints in Fig. 2. So, the basic checkpointing algorithm will not work in the absence of sufficient number of application messages that help each process know the status of every other process in a timely manner. We call this as a consistent global checkpoint convergence problem and explain in Section 4.5.1 how it can be addressed by using limited number of control messages when necessary. Next, we introduce the data structures needed for presenting the basic algorithm.

4.3. Data structures

Each process $P_i$ maintains the following data structures.

1. $csn_i$: 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.

2. $stat_i$: 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.

3. $logSet_i$: The set of messages logged at $P_i$ after it takes a tentative checkpoint. When $stat_i$ is set to tentative, $P_i$ sets $logSet_i$ to empty and starts logging messages sent and received by $P_i$ into $logSet_i$. Thus, $logSet_i$ contains messages sent and received by $P_i$ after
a tentative checkpoint is taken and before that checkpoint is finalised. When the status of the process changes from tentative to normal, the tentative checkpoint and the corresponding logSet are flushed to the stable storage.

4. \(P_i\) tentative process set maintained at \(P_i\). When stat, is set to normal, \(P_i\) tentative set is set to empty. When \(P_i\) takes a tentative checkpoint, \(P_i\) sets \(P_i\), to \([P_i]\). Upon receiving a message, \(P_i\) sets \(P_i\), to be the union of \(P_i\), and the tentative process set piggy-backed in the message. Thus, this set contains the set of processes that have taken a tentative checkpoint, to the knowledge of \(P_i\).

5. \(allPSet\): This is the set of all processes, namely, \([P_0, P_1, \ldots, P_{n-1}]\).

4.4. 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.

4.4.1. Consistent global checkpointing initiation

Any process whose status is normal can take a new tentative checkpoint, thereby initiating consistent global checkpointing. When a process \(P_i\) takes a tentative checkpoint, it changes its status from normal to tentative, increases the checkpoint sequence number \(csn\) by one and assigns it as the sequence number for the tentative checkpoint, sets logSet, to empty, and initializes tentSet, to \([P_i]\). At any time, tentSet, is the set of all processes that have taken a tentative checkpoint corresponding to the current tentative checkpoint of \(P_i\), to the knowledge of \(P_i\). After \(P_i\) takes a tentative checkpoint, it starts logging into logSet, all the messages sent and received until its status changes back to normal. Csn and tentSet, are piggy-backed with each application message.

4.4.2. Sending messages

Each process \(P_i\) piggy-backs with each application message the current value of \(csn\), stat, and tentSet. The value of \(csn\), piggy-backed with messages, helps the receiver determine if the sender took a new tentative checkpoint, thereby initiating a concurrent or new consistent global checkpoint collection. These values piggy-backed with a message \(M\) are denoted by \(M.csn\), \(M.stat\) and \(M.tentSet\) respectively. A process receiving message \(M\) uses this piggy-backed information to find out whether it is a new consistent global checkpoint collection initiation or a concurrent global checkpoint initiation; it also comes to know the processes that have already taken a tentative checkpoint corresponding to this initiation.

4.4.3. Receiving messages

Under our algorithm, each process can take a tentative checkpoint independently and concurrently. Once a process comes to know that all the other processes have taken tentative checkpoints corresponding to its most recent tentative checkpoint (through a message received from a process), it finalizes the tentative checkpoint (Section 4.4.4 explains the procedure of finalizing a tentative checkpoint). After finalizing its most recent tentative checkpoint \(C_{i,k}\), process \(P_i\) can take the next tentative checkpoint \(C_{i,k+1}\) before every other process has finalized the tentative checkpoint corresponding to \(C_{i,k}\). In such situations, if \(P_i\) sends a message \(M\) after taking \(C_{i,k+1}\) and \(M\) is received by process \(P_j\) before it finalized \(C_{j,k}\), then \(P_i\) needs to finalize \(C_{j,k}\) first and then process the message \(M\) to prevent orphan messages. Next, we describe how process \(P_i\) handles a message \(M\) received from process \(P_j\).

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

Case (2) \(M.stat = stat_i = tentative\). In this case, both \(P_i\) and \(P_j\) have taken a new tentative checkpoint concurrently. The following four subcases arise:

Subcase (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 \(M\) and \(P_j\) was not aware of this while sending \(M\). Therefore, no additional action needs to be taken except processing the message.

Subcase (b) \(M.csn = csn_i\). In this case, \(P_i\) and \(P_j\) have taken checkpoints that belong to the same global checkpoint \(S_{csn}\). In this case, \(P_i\) is processed and then in order to know how many processes have taken a tentative checkpoint that belongs to the global checkpoint \(S_{csn}\), \(P_i\) updates tentSet, to be the union of tentSet, and \(M.tentSet\). If the updated tentSet, equals to allPSet, \(P_i\) logs the message and then finalizes (Section 4.4.4 gives the detailed procedure for finalizing a tentative checkpoint) 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 \(S_{csn}\), and sets its status to normal \(i.e.,\) stat, = normal).

Subcase (c) \(M.csn = csn_i + 1\). In this case, \(P_i\) finalized the checkpoint with sequence number \(csn_i\) before sending \(M\) 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 \(S_{csn}\). Recall that a process is not allowed to take a new tentative checkpoint until it has finalized its current tentative checkpoint. Thus, \(P_i\) finalizes its current tentative checkpoint with sequence number \(csn_i\) without including \(M\) in the message log because \(M\) would be an orphan message with respect to the consistent global checkpoint \(S_{csn}\). Then, it processes the message \(M\) and initiates next consistent global checkpointing by taking a new tentative checkpoint with sequence number \(M.csn\) and also logs the message \(M\).

Subcase (d) \(M.csn > csn_i + 1\). In this case, \(P_i\) has finalized the checkpoint with sequence number \(csn_i + 1\). Since \(P_i\) could have finalized that checkpoint only after 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. Thus, this case is not shown in the formal description of the algorithm.

Case (3) \(M.stat = normal\) and \(stat_i = tentative\). In this case, \(P_i\)’s latest checkpoint has been finalized before sending \(M\) and \(P_i\) has taken a tentative checkpoint which is yet to be finalized. The following three subcases arise:

Subcase (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 except processing the message.
Subcase (b) \( M , csn = csn_i \). In this case, \( P_i \) has finalized taking the checkpoint with sequence number \( csn_i \). This means \( P_i \) knows that all processes have taken a tentative checkpoint with sequence number \( csn_i \). Hence \( P_i \) finalizes its current tentative checkpoint without including \( M \) in the message log (since \( M \) would be an orphan message), changes its status back to normal and then process the message.

Subcase (c) \( M , csn > csn_i \). This means \( P_i \) 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 \( M , csn \) before other processes finalize their checkpoint with sequence number \( M , csn - 1 \). So, this case does not arise.

Case (4) \( M , stat = tentavite \) and \( stat_i = normal \). This means \( P_i \)’s latest checkpoint taken before sending \( M \) has not been finalized while sending \( M \) and \( P_i \)’s latest checkpoint has been finalized. In this case, \( M \) is processed first and then the following actions are taken. The following three subcases arise:

Subcase (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.

Subcase (b) \( M , csn = csn_i + 1 \). In this case, \( P_i \) has taken a new tentative checkpoint about which \( P_i \) comes to know through \( M \). 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 4.4.1. In addition to that, \( P_i \) logs the message and updates \( tentSet_i \) to be the union of \( tentSet_i (= \{ | P_i \}) \) and \( M , tentSet \). Thus, \( P_i \) gets \( P_i \)’s knowledge about the processes that have taken a tentative checkpoint with sequence number \( csn_i + 1 \).

Subcase (c) \( M , csn > csn_i + 1 \). This case is similar to subcase (d) under case (2) and does not arise.

4.5. Optimizations

4.5.1. A convergence problem

As we noted earlier, the basic checkpointing algorithm presented in the previous section may not converge if not enough messages are exchanged among processes. 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 type of control messages — checkpoint begin (\( CK_BGN \)), 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, say \( P_j \). Upon receiving the message, \( P_j \) takes a tentative checkpoint if it has not yet taken and then sends a \( CK_REQ \) message to \( P_i \). \( P_i \) does the same and sends it to \( P_j \), etc. and finally \( CK_REQ \) reaches back to \( P_i \). After \( P_i \) 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 tentative 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 \( P_i \) 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 \( tentSet \), and \( j \) is less than \( i \). If \( P_j \) exists, \( P_i \) does nothing since it knows that \( P_i \) 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 process 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) Reducing \( CK_REQ \) messages. Under the above approach, every process needs to forward the \( CK_REQ \) message 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 \( P_k \) for which the following condition holds.

\[
(j > i) \text{ AND } (P_i \not\in tentSet) \text{ AND } (\forall k \in \{ | i < j \}), \ P_k \in tentSet.
\]

If such a process \( P_j \) is found, \( P_i \) forwards the message to \( P_j \) because all processes with process ids 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 ids greater than \( i \) have already taken a tentative checkpoint. Therefore, \( P_i \) forwards the message to \( P_0 \) directly.

4.4.4. 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 any process in the system is tentative (i.e. \( tentSet_i = allPSet \)), it flushes its current tentative checkpoint (the most recent tentative checkpoint taken), if it has not already done so, and also the associated message log \( logSet_i \) into the stable storage and makes it permanent. Note that the tentative checkpoint can be flushed to stable storage any time before finalizing the tentative checkpoint. However, the message log associated with the tentative checkpoint needs to be flushed as soon as a process comes to know that all other processes have taken a tentative checkpoint corresponding to its latest checkpoint. 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. Checkpoints with same sequence number from all the processes form a consistent global checkpoint, as proved in Theorem 2.

Formal description of the basic checkpointing algorithm is given in Fig. 3.
Fig. 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(type, csn) refers to the control message CM with CM.type = type and CM.csn = csn. For example, CM(CK_BGN, 3) refers to a control message CK_BGN with csn = 3 piggy-backed 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 an example shown in Fig. 5. Suppose P1 takes a tentative checkpoint CT1,1 first and sends a message M2 to P2. Upon receiving M2, P2 takes a tentative checkpoint CT2,1. When the timer set for CT1,1 expires, P1 sends a CK_BGN message (CK_BGN1) to P0 (P2 does not send a CK_BGN message since it knows that P1 will send such message to P0). Upon receiving CK_BGN1, P0 takes a tentative checkpoint CT0,1 and sends a CK_REQ message CK_REQ1 to P1. Thereafter, P1 sends a CK_REQ message CK_REQ2 to P3 since it knows that P3 has already taken CT3,1. Finally, the CK_REQ message CK_REQ3 returns to P0. Now, P0 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 CT0,1 to the stable storage. Upon receiving CK_END, P1, P2 and P3 flush their logged messages and tentative checkpoints with sequence number 1 respectively.

When P1 starts
\[ \text{csn}_1 = 0; \quad \text{stat}_1 = \text{normal}; \]

Procedure: takeTentativeCheckpoint(i: integer)
\[ \text{csn}_i = \text{csn}_i + 1; \quad \text{stat}_i = \text{tentative}; \]
\[ \text{tentSet}_i = \{ P_i \}; \]
\[ \text{logSet}_i = \emptyset; \]

Take tentative checkpoint CT1,csn1;

When P1 starts to take a checkpoint

\[ \text{takeTentativeCheckpoint}(i); \]

When P1 sends a message M to Pj
\[ M; \text{csn} = \text{csn}_i; \]
\[ M; \text{stat} = \text{stat}_i; \]
\[ M; \text{tentSet} = \text{tentSet}_i; \]
if \[ \text{stat}_i == \text{tentative} \]
then \[ \text{logSet}_i = \text{logSet}_i \cup \{ M \}; \]
Send(M);

When P1 receives a message M from Pj
\[ \text{if} \quad \text{stat}_i == \text{normal} \]
\[ \text{if} \quad \text{M;stat} == \text{tentative} \]
Process M;
\[ \text{if} \quad \text{M;csn} == \text{csn}_i + 1 \]
\[ \text{takeTentativeCheckpoint}(i); \]
\[ \text{logSet}_i = \text{logSet}_i \cup \{ M \}; \]
\[ \text{tentSet}_i = M; \text{tentSet} \cup \text{tentSet}_i; \]
else if \[ \text{stat}_i == \text{tentative} \]
\[ \text{logSet}_i = \text{logSet}_i \cup \{ M \}; \]
\[ \text{if} \quad \text{M;stat} == \text{normal} \]
\[ \text{if} \quad \text{M;csn} == \text{csn}_i \]
Flush \[ \text{logSet}_i \cup \{ M \} \]
\[ \text{stat}_i = \text{normal}; \]
Process M;
else if \[ \text{M;stat} == \text{tentative} \]
\[ \text{if} \quad \text{M;csn} == \text{csn}_i \]
Process M;
\[ \text{tentSet}_i = M; \text{tentSet} \cup \text{tentSet}_i; \]
\[ \text{if} \quad \text{tentSet}_i == \text{allPS} \]
\[ \text{stat}_i = \text{normal}; \]
Flush \[ \text{logSet}_i \]
\[ \text{else if} \quad \text{M;csn} == \text{csn}_i + 1 \]
\[ \text{P1 has taken the checkpoint CT1,csn1 before sending the message} \]
\[ \text{P1 has initiated a new consistent global checkpoint} \]
\[ \text{P1 has finalized the checkpoint CT1,csn1} \]
\[ \text{P1 finalizes its checkpoint CT1,csn1} \]
\[ \text{P1 has initiated a new consistent global checkpoint} \]
\[ \text{P1 has finalized the checkpoint CT1,csn1} \]
\[ \text{P1 finalizes its checkpoint CT1,csn1} \]

Fig. 3. The basic checkpointing algorithm.
When the timer for finalizing the tentative checkpoint on $P_i$ expires

1. If $i = 0$, then
   - forwardCheckpointRequest($P_0, CM$);
   
   /* $P_0$ initiates CK.REQ messages directly without sending a CK.BGN message */

   else
   
   for each $P_k \in \text{tentSet}_i$, do
   
   if $k < i$ then return;
   
   Send $CM(\text{CK.BGN, csn}_i)$ to $P_0$;
   
   /* $P_k$ or other process with process number less than $k$ will send CK.BGN message to $P_0$ */
   /* Sending CK.BGN message to $P_0$ */

   Procedure: forwardCheckpointRequest($P_i, CM$)

   if $i = N - 1$ then $k = 0$;
   
   else
   
   for $k = i + 1$ to $N - 1$ do
   
   if $P_k \notin \text{tentSet}_i$, then break;
   
   if $P_k \in \text{tentSet}_i$, then $k = 0$;
   
   Send $CM(\text{CK.REQ, csn}_i)$ to $P_k$;
   
   /* The status of all processes with process number greater than $i$ is tentative */

When $P_i$ receives $CM$ from $P_j$

1. If $CM.csn == csn_i + 1$ then
   
   if $stat_i == \text{tentative}$ then
   
   Flush logSet$_i$ and $CT_i,csn_i$ to the stable storage;
   
   takeTentativeCheckpoint(i);
   
   forwardCheckpointRequest($P_i, CM$);
   
   else if $CM.csn == csn_j$, then
   
   if $CM.type == \text{CK.BGN}$ then
   
   if $CM(csn_i)$ has been sent return;
   
   forwardCheckpointRequest($P_i, CM$);
   
   else if $CM(\text{CK.END, csn}_i)$ has not been sent then
   
   Send $CM(\text{CK.END, csn}_i)$ to $P_1, P_2, \cdots, P_{N-1}$;
   
   else if $CM.type == \text{CK.REQ}$ then
   
   if $i == 0$ then
   
   if $CM(\text{CK.END, csn}_i)$ has been sent return;
   
   Send $CM(\text{CK.END, csn}_i)$ to $P_1, P_2, \cdots, P_{N-1}$;
   
   if $stat_i == \text{tentative}$ then
   
   $stat_i$ = normal;
   
   Flush logSet$_i$ and $CT_i,csn_i$ to the stable storage;
   
   else forwardCheckpointRequest($P_i, CM$);
   
   else if $stat_i == \text{tentative}$ then
   
   $stat_i$ = normal;
   
   Flush logSet$_i$ and $CT_i,csn_i$ to the stable storage;

   /* $CM.type == \text{CK.END} */

4.6. 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 converges, i.e. after a process takes a tentative checkpoint with a given sequence

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 $P_3$ cannot finalize its tentative checkpoint $CT_{3,1}$ without the help of control messages.
number $c_{sn}$, every process eventually finalizes a checkpoint with sequence number $c_{sn}$.

**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 took a tentative checkpoint $CT_{i,k}$ but never finalized 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 $CM(C_K,REQ,k)$ from a process $P_j$. Upon receiving such a message, $P_i$ needs to forward the message to a process $P_k$ 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 $P_0$ ($P_0$ raises the message to $P_0$ or some process $P_j$ ($j < N - 1$) forwards it to $P_0$ directly since $P_i$ knows that all processes with process number greater than $j$ have already taken a tentative checkpoint with sequence number $k$). Once $P_0$ receives the message, it finalizes $C_0,k$ and broadcasts a message $CM(C_K,END,k)$ to all other processes. Upon receiving this message, each process finalizes its tentative checkpoint with sequence number $k$ if it has not already done so. 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 $k$, $P_i$ has initiated a message $CM(C_K,REQ,k)$. Otherwise, $P_i$ or some process with process number smaller than $i$ will send a message $CM(C_K,BGN,k)$ to $P_0$. Therefore, $P_0$ will receive at least one $C_K,BGN$ message with sequence number $k$. Then $P_0$ initiates the process or 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 $S_k = \{C_{i,k} | i \in 0, 1, \ldots, N - 1\}$ is 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 $i, j \in 0, 1, \ldots, N - 1, i \neq j$), such that $C_{i,k} \xrightarrow{hb} send(M)$ AND $receive(M) \xrightarrow{hb} C_{i,k}$.

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

**Case (1)** receive($M$) $\xrightarrow{hb}$ $CT_{i,k}$ (a). Since $C_{i,k} \xrightarrow{hb} send(M)$, $CFE_{i,k} \xrightarrow{hb} send(M)$ (b). Since $P_i$ has finalized $C_{i,k}$, $P_j$ has known that each process $P_i$ has taken tentative checkpoint $CT_{j,k}$. Therefore, $CT_{j,k} \xrightarrow{hb} CFE_{j,k}$ (c). From (a), (b) and (c), we have $receive(M) \xrightarrow{hb} CT_{j,k} \xrightarrow{hb} CFE_{j,k} \xrightarrow{hb} send(M)$, i.e. $receive(M) \xrightarrow{hb} send(M)$, a contradiction.

**Case (2)** $CT_{i,k} \xrightarrow{hb} receive(M) \xrightarrow{hb} CFE_{i,k}$ (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, $P_i$ finalizes the checkpoint $C_{i,k}$, not including message $M$ in the checkpoint. Therefore, we have $CFE_{i,k} \xrightarrow{hb} receive(M)$ (b). From (a) and (b) we have $receive(M) \xrightarrow{hb} receive(M)$ which is a contradiction.

Hence the theorem. □

### 4.7. Recovery algorithm

In this section, we present a recovery algorithm based on the checkpointing algorithm. We make the following assumption for the recovery algorithm.

- At most one process fails at any given time. No other process fails until the recovery due to a failed process is complete.

We need to add the following data structures to the checkpointing algorithm presented in Sections 4.4 and 4.5.

- Each process $P_i$ has a variable $rsn_i$, initialized to 0, to keep track of the total number of times recovery took place. Each time $P_i$ initiates recovery, this variable is incremented by 1.

**Informal description of the recovery algorithm**

When a process $P_i$ fails, it increments $rsn_i$ by 1 and sends $ROLLBACK(rsn_i, c_{sn})$ message to all the processes; here $c_{sn}$ represents the sequence number of the latest finalized checkpoint of the process $P_i$. When a process $P_j$ receives $ROLLBACK(rsn_i, c_{sn})$ message from process $P_i$, it finalizes the checkpoint with sequence number $c_{sn}$ if it has not already done so, and then sends $OKTOROLLBACK(rsn_i, c_{sn})$ to $P_i$. After a process sends $OKTOROLLBACK$ message, it blocks (i.e., it neither sends/receives any application message nor does any local computation). After $P_i$ receives $OKTOROLLBACK$ reply from all the processes, it sends $CONFIRMROLLBACK(rsn_i, c_{sn})$ message. After a process $P_i$ receives $CONFIRMROLLBACK(rsn_i, c_{sn})$ message, it retrieves the finalized checkpoint $C$ with sequence number $c_{sn}$, rolls back to the tentative checkpoint with sequence number $c_{sn}$ stored in $C$, and replays the messages in the log associated with $C$ and then sends $ROLLBACKFINISHED(rsn_i, c_{sn})$ message to $P_i$ and blocks. After $P_i$ receives $ROLLBACKFINISHED(rsn_i, c_{sn})$ from all processes, it sends $PROCEED(rsn_i, c_{sn})$ message to all the processes. Upon receiving the $PROCEED$ message, each process resumes its computation normally.

**Formal description of the recovery algorithm**

A process finalizes its tentative checkpoint with a given sequence number only after it comes to know that all the other processes have taken their tentative checkpoints with the same sequence number. When a process fails, all processes roll back to the checkpoint with the same sequence number. Note that a checkpoint of a process consists of the saved state of the process (tentative checkpoint) and the log of messages sent and received after the tentative checkpoint was taken and before the tentative checkpoint was finalized. The fact that the checkpoints of all the processes with the same sequence number forms a consistent global checkpoint has been proved in Section 4.6. Thus rolling back the processes to checkpoints with same sequence number takes the state of the processes to a state represented by a consistent global checkpoint. However, messages lost due to rollback such as those whose receive event was undone while the corresponding send event has not been undone are not taken care of. They can be handled using sequence number and message logging. Moreover, we do not discuss ways for handling concurrent failures. However, methods similar to the ones used in [22] can be used for handling concurrent failures as well as handling lost messages, duplicate messages in-transit messages during recovery.
When $P_i$ fails and initiates recovery process
\[
rsn_i = rsn_i + 1;
\]

Sends $ROLLBACK(rsn_i, csn_i)$ to all processes; // $csn_i$ is the sequence number of the latest finalized checkpoint of $P_i$;

When $P_j$ receives $ROLLBACK(rsn_i, csn_i)$ from $P_i$

if $rsn_j < rsn_i$ then // this is a new recovery initiation
\[
rsn_j = rsn_i;
\]

Finalizes the tentative checkpoint with sequence number $csn_i$
if it has not already done so;
Sends $OKTOROLLBACK(rsn_i, csn_i)$ reply to $P_i$;
Blocks;

After $P_i$ receives $OKTOROLLBACK(rsn_i, csn_i)$ from all processes

Sends $CONFIRMROLLBACK(rsn_i, csn_i)$ to all processes;

When $P_j$ receives $CONFIRMROLLBACK(rsn_i, csn_i)$ from $P_i$

Finds the finalized checkpoint $C$ with sequence number $csn_i$;
Rolls back to the tentative checkpoint contained in $C$;
Replays the messages in the message log associated with $C$;
Sends $ROLLBACKFINISHED(rsn_i, csn_i)$ to $P_i$;
Blocks;

After $P_i$ receives $ROLLBACKFINISHED(rsn_i, csn_i)$ from all processes:

Sends $PROCEED(rsn_i, csn_i)$ to all processes;

When $P_j$ receives $PROCEED(rsn_i, csn_i)$

$P_j$ resumes computation;

Fig. 6. Recovery algorithm.

5. Performance evaluation

In this section, we present the performance evaluation of our algorithm. We denote our algorithm as OCML (Optimistic Checkpointing and Message Logging approach) for short. We evaluated our algorithm with respect to the following two aspects: (1) under what scenarios our algorithm converges without using additional control messages and what is the overhead induced by the control messages; (2) how does it perform compared to Vaidya’s algorithm [29], which we refer to as Vaidya_Stagger. The comparison focuses on the latency and network contention for stable storage.

5.1. Simulation model

We consider distributed computations running in an environment that has the following features:

- **Network environment.** All processes run on nodes in a local area network (LAN). We assume that the average end-to-end message delay is 5 ms.
- **Clock drift.** We assume that the maximum drift of local clocks at various sites is 100 ms per h.
- **Simulation time.** It is set to 100 min.
- **Checkpoint initiation.** We divide the simulation time into 10-min intervals. These intervals are called checkpoint intervals. Thus, each process has 10 checkpoint intervals during its life time. Each process chooses the time to take tentative checkpoints randomly in each interval. When control messages are used for convergence, we set the value of timeout for finalizing a checkpoint to be 5 min. That is, a process initiates sending control messages if it does not finalize its tentative checkpoint in 5 min.
- **Communication model.** We simulated under two types of Checkpoint and Communication Patterns (CCPAT), namely, RANDOM and GROUP, described below:

  - **RANDOM Communication Pattern:** Each process $P_i \in P_0, P_1, \ldots, P_{N-1}$ is able to send an application message to any other process $P_j \in P_0, P_1, \ldots, P_{N-1}$ and $P_i \neq P_j$. The destination of each message $m$ is randomly chosen. Messages sent are uniformly distributed during the entire simulation time of a process.

  - **GROUP Communication Pattern:** Each process $P_i \in P_0, P_1, \ldots, P_{N-1}$ sends/receives messages only to/from its two neighbor processes $P_{(i-1) \mod N}$ and $P_{(i+1) \mod N}$. This basically means that processes are logically arranged in a ring and each process sends messages only to its two neighbors.

  We choose these two CCPATs mainly because they are representatives of many long-running, compute-intensive applications [10]. For example, in the implementation of Gaussian elimination, in each iteration, a process receives a row of the matrix from its predecessor and sends the results of its computation to its successor. Its communication model among processes is fits into our GROUP Communication Pattern. Moreover, these two models have been regarded as two extreme representatives for distributed applications [6]. So we ran our simulations under these two extreme models to evaluate the performance of our algorithm. In all the simulation runs, we varied the rate of messages sent per second by each process from 0.01 to 0.40, on average. Our goal is to study not only the number of control messages needed under sparse communication pattern but also the network contention for stable storage under dense communication pattern.

5.2. Simulation results

In this section, we first present our simulation results regarding (i) under what scenarios our algorithm converges without using additional control messages and (ii) what is the overhead induced by the control messages. We also evaluate the number of messages logged for the purpose of determining consistent global checkpoint. Then we compare the performance of our algorithm with the algorithm of Vaidya.
1. OCML with control messages vs. OCML without control messages

We evaluated the performance of our algorithm with control messages and without control messages under the RANDOM communication model. We simulated a distributed computation involving 20 processes. Fig. 7(a) shows the number of finalized global checkpoints for various message patterns. Ideally, our algorithm should take 10 consistent global checkpoints since the simulation time is 100 min and the checkpoint interval is 10 min. Irrespective of the rate at which messages are exchanged, our algorithm takes exactly 10 consistent global checkpoints if control messages are used. This verifies that the use of control messages helps in convergence, especially when application messages are exchanged at a low rate. However, without control messages, only 6 consistent global checkpoints are finalized if each process sends only 0.01 messages per second. This means that processes have to wait for a long time for finalizing a checkpoint. As the rate of messages sent per second by each process increases, our algorithm converges quickly; it only requires 0.03 messages or more per second to converge without any control messages.

Fig. 7(b) shows the average amount of time (in seconds) needed for taking a consistent global checkpoint, this time being calculated from the time some process initiates consistent global checkpointing to the time at which all processes finalize their tentative checkpoints belonging to this global checkpoint. The average time for taking a consistent global checkpoint is a little more than 300 s if less than 0.05 messages are sent by each process per second, in which case control messages are used. If more than 0.05 messages are sent by each process per second, processes finalize their tentative checkpoints before the timer expires. Therefore, no control messages is sent in this case. Fig. 7(c) verifies this observation. We also note that the number of control messages sent are less than 2 times the number of processes even when only 0.01 messages are sent by each process per second.

Fig. 7(d) shows the number of logged messages for each global checkpoint at each process. In the figure, the number of logged messages for the case when no control message is sent does not change much as the rate of messages sent per second by each process increases. This also reveals the approximate number of messages needed for the convergence of our algorithm under this communication model. Since the logged messages contain messages sent and received at each process, our algorithm requires each process send only 6 to 9 messages per checkpoint interval for it to converge when 20 processes are involved.

2. Performance of our algorithm compared to Vaidya’s algorithm

Next, we present the performance analysis of our algorithm (denoted as OCML) compared to Vaidya’s staggered checkpointing algorithm [29] (denoted as Vaidya_Stagger) in this section. We choose Vaidya’s algorithm mainly because (1) it represents the staggered checkpointing algorithms which attempt to prevent two or more processes take checkpoints at the same time in order to reduce contention for stable storage; (2) to our knowledge, it is the only algorithm that tries to stagger checkpoints to prevent contention for accessing stable storage; (3) moreover, Vaidya’s notion of “physical checkpoint + message log = logical checkpoint” [29], is similar to our notion of “tentative checkpoints + message log = finalized checkpoint”.

We compared the performance of our algorithm with Vaidya’s algorithm [29], under both RANDOM and GROUP communication models.

First, we compare our algorithm with Vaidya’s algorithm with respect to the average number of checkpoints (note
here checkpoints refer to physical checkpoints under Vaidya’s algorithm and tentative checkpoints under our algorithm respectively) taken at the same time by each process. Table 1 shows the results as the rate of messages sent per second by each process varies from 0.01 to 0.10. Since Vaidya’s algorithm successfully staggers all physical checkpoints, the average number of physical checkpoints taken at the same time under all cases for this algorithm are zero. However, this goal has been achieved at the cost of large increase in checkpoint latency in Vaidya’s algorithm [29]. On the other hand, although the average number of tentative checkpoints taken at the same time in our algorithm is not zero, since each process is able to store the tentative checkpoint in memory first and choose its convenient time for writing the tentative checkpoints to stable storage at the network file server, it doesn’t incur any contention for stable storage in the tentative checkpointing phase of our algorithm while at the same time decreasing the checkpoint latency.

Next, we compare the performance of our algorithm with Vaidya_Stagger with respect to the number of logged messages under both RANDOM and GROUP communication models. Under the RANDOM communication model, Fig. 7(d) shows the number of logged messages under OCML with CtrlMessages and OCML without CtrlMessages. Fig. 8(a) and (c) show the performance results of our algorithm compared to Vaidya_Stagger under RANDOM and GROUP communication models respectively, as the rate of messages sent per second by each process varies from 0.02 to 0.20. Fig. 8(d) shows the result under GROUP model, as the rate of messages sent per second by each process varies from 0.20 to 0.40. As expected, under both communication models, when the rate of messages sent per second by each process increases, our algorithm converges fast and doesn’t need control messages. Under RANDOM model, as the rate of messages sent per second by each process increases, the number of logged messages in our algorithm is always smaller than that of Vaidya_Stagger. Under the GROUP communication model, the number of logged messages under our algorithm continues to be smaller than that of Vaidya_Stagger if the rate of messages sent per second by each process is larger than 0.08. Fig. 8(b) shows how the number of logged messages changes with respect to the number of processes involved in the computation under RANDOM model. The results indicate a linear increase in the number of logged messages in Vaidya_Stagger with respect to the number of processes. On the other hand, increase in the number of processes has only slight impact on the number of logged messages in our algorithm, which indicates that our algorithm is more scalable.

Finally, under both RANDOM and GROUP communication models, we compare our algorithm and Vaidya_Stagger with respect to the contention for stable storage at the network file server that arises due to storing logged messages. Fig. 9(a) and (c) show the results under RANDOM and GROUP communication models respectively, as the rate of messages sent per second by each process varies from 0.02 to 0.20. Fig. 9(d) shows the result under GROUP communication model, as the rate of messages sent per second by each process varies from 0.20 to 0.40. Since in the second phase of Vaidya_Stagger, each process takes its logical checkpoint by logging messages on stable storage after receiving the marker message from the coordinator, it means that the coordinator plays the centralized role of synchronizing the message-logging in each process and it may lead to a single point of failure. It completely staggers the physical checkpoints, however, contention for access to stable storage still occurs while storing logged messages [29]. As a result, the number of collisions due to logged messages in each process is the same as the number of logical checkpoints taken at each process in Vaidya_Stagger. However, in our algorithm, under the RANDOM model, Fig. 9(a) shows the average number of collisions due to logged messages is 3.6 without CtrlMessage, which is 64% less than that of Vaidya_Stagger. Under the GROUP communication model, as shown in Fig. 9(d), as the rate of messages sent by each process varies from 0.21 to 0.40 per s, the average number of collisions due to logged message is 6.3 for both OCML with CtrlMessages and OCML without CtrlMessages, which is 37% less than that of Vaidya_Stagger. Fig. 9(b) shows how the number of collisions due to logged messages changes with respect to the number of processes involved in the computation under RANDOM model. As expected, when the number of processes increases, the number of collisions due to logged messages under our algorithm only has slight impact and it is at least 60% less than that of Vaidya’s algorithm. Vaidya’s algorithm [29] successfully staggers all physical checkpoints so that no contention for stable storage occurs while storing physical checkpoints. However, it does incur contention for stable storage when messages are logged in its second phase. Compared to Vaidya_Stagger, although the average number of tentative checkpoints taken at the same time under our algorithm is not zero, it doesn’t incur any contention for stable storage since each process is able to store the tentative checkpoint in memory first and choose its convenient time for writing the tentative checkpoints to stable storage at the network file server. For example, based on our simulation results, we can choose to save the tentative checkpoint together with its corresponding logged messages at the same time when it is finalized or earlier when there is no contention for stable storage. In minimizing contention for stable storage at the network file server, our algorithm always performs better than Vaidya_Stagger. And our algorithm also has other desirable features such as low control messages (or even no control messages) and less checkpoint latency compared to Vaidya_Stagger algorithm. Moreover, our algorithm is distributed whereas Vaidya’s algorithm is centralized.

6. Conclusion

In this paper, we presented a novel communication-induced checkpointing algorithm that makes every checkpoint belong to a consistent global checkpoint. Under this algorithm, every process stores the tentative checkpoint in memory first and then flushes it to stable storage when there is no contention for stable storage or after finalizing the tentative checkpoint. Messages sent and received after a process takes a tentative checkpoint are logged into memory until the tentative checkpoint is finalized. Since a tentative checkpoint can be flushed to stable storage any time before finalizing it, contention for stable network storage that arises due to several processes storing the checkpoints simultaneously is reduced/eliminated. Moreover, unlike existing communication-induced checkpointing algorithms, our algorithm, in general, 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 the checkpoint. This improves the response time for messages. It also helps a process take the regularly scheduled basic checkpoints at those times. If messages are not frequently exchanged among processes, additional control messages may be required for the algorithm to collect consistent global checkpoints in a timely manner. We augmented the basic algorithm with control messages to speed up the collection of consistent global checkpoints in a timely manner for applications in which processes do not communicate frequently. We conducted a performance evaluation of the algorithm and studied the
Fig. 8. Number of logged messages under OCML and Vaidya_Stagger.

Fig. 9. Number of collisions due to storing logged messages at the network file server under OCML and Vaidya_Stagger.
overhead induced by the control messages which also helps in determining when control messages are needed. We also compared the performance of our algorithm with Vaidya’s algorithm [29]. In minimizing the contention for stable storage at the network file server, our algorithm always performs better than Vaidya’s algorithm. Our algorithm also has other desirable features such as the scalability, low control messages (or even no control messages) and less checkpoint latency compared to Vaidya’s algorithm.

Acknowledgments

The authors thank the editor and the reviewers for their valuable and constructive comments which helped greatly in improving the content and presentation of the paper. Preliminary version of this paper has been presented in IEEE International Parallel and Distributed Processing Symposium 2007 [11]. 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


Qiangfeng Jiang is a Ph.D. candidate in the department of Computer Science at University of Kentucky. He received his B.S and M.S. in Computer Science from Tongji University, Shanghai, China in 1998 and East-China Institute of Computer Technology, Shanghai, China in 2001 respectively. His current research interests lie in the areas of wireless networks, channel allocation, and checkpointing and recovery in distributed systems.
Yi Luo received both her B.S. and M.S. degrees in computer science from University of Electronic Science and Technology of China, Chengdu, China. She is currently pursuing her Ph.D. degree in the Department of Computer Science, University of Kentucky, Lexington, Kentucky, USA. Her research interests include power-aware routing and energy efficient routing protocols for mobile ad hoc networks, wireless sensor networks and fault-tolerance in distributed systems.

D. Manivannan is currently an associate professor in the Department of Computer Science at the University of Kentucky, Lexington, Kentucky, USA. Dr. Manivannan received the Ph.D. degree in computer and information science from The Ohio State University, Columbus, Ohio, in 1997. His areas of research include fault-tolerance and synchronization in distributed systems, routing in wormhole networks, channel allocation in cellular networks, routing in ad-hoc networks and wireless personal area networks. He served as program committee member for several International Conferences and also served as reviewer for several International Journals published by ACM, IEEE, Elsevier, Springer Verlag, Oxford University Press and others. He is on the Editorial board of Wireless Personal Communications Journal, published by Kluwer Publishers. He is also serving as an Associate Technical Editor for IEEE Communications Magazine. Information about his publications and research can be found at his lab site http://dmcslab.netlab.uky.edu/. Dr. Manivannan is a recipient of the CAREER Award from the US National Science Foundation. He is a senior member of both IEEE, and ACM.