Distributed System Project
Report

Synchronous Checkpointing and Recovery

Submitted by

Roll No	Names of Students
12CS10002	Aayush Singhal
12CS10042	Sabyasachi Behera
12CS30016	Bhushan Kulkarni
12CS30028	Rohit Agrawal

Under the guidance of
Arobinda Gupta

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Kharagpur, West Bengal, India – 721 302

Spring Semester 2016

Contents

Chapter 1

Introduction

1.1 Problem Definition

Today there are lots of uses of distributed systems including client-server system, World Wide Web and many more. They are not fault-tolerant and its vast use make it susceptible to failures. Therefore many techniques have been proposed to make it more reliable and to increase its availability. In this paper, we will discuss about one of its solution i.e, checkpointing and their types. We will also discuss various algorithms associated with synchronous checkpointing.

1.2 General Discussion

A distributed system consists of fixed number of processes which does not have any shared memory. Their only source of communication is through messages. They also communicate through outside world through input and output messages.

In this paper, we will discuss about the rollback recovery protocols which restores the system back to a consistent state after a failure. It achieves fault tolerance by periodically saving the state of all the process during fault-free execution which enables it to start from some previously saved state. This saved state are called checkpoints and the complete process of recovering checkpoint is known as rollback recovery. This checkpoints can be saved either is stable storage or volatile storage depending on the failure scenarios to be tolerate.

1.3 Issues in failure recovery

In a failure recovery, we must not only restore the system to a consistent state but we also have to handle different kind of messages that are left in an abnormal state. Different kind of messages are described below:

In-transit message

The messages which has been sent but not yet received. When in-transit messages are part of global system state, they do not cause any inconsistency. However, depending on whether a system assumes reliable communication channels, rollback recovery protocols may have to guarantee the delivery of in-transit message.

Lost messages

Message whose send is not undone but receive is undone due to rollback are called lost messages. This type of messages occur when the process rolls back to a checkpoint prior to the reception of a message while sender does not rollback beyond the send operation of the message.

Delayed messages

Message whose receive is not recorded because the receving process was either down or the message arrived after the rollback of the receiving process are called delayed messages.

Orphan messages

Messages with receive recorded but message send not recorded are called orphan messages. For example, a rollback might have undone the send of an message but does not undone the receive of that message. This type of message does not arise if we rollback to a global consistent state.

Duplicate message

It arise due to message logging and replaying during process recovery.

Hence, we have to handle all this kind of messages during the rollback recovery and handling this kind of messages is the biggest challenge in this recovering algorithms.

1.4 Types of Checkpointing

Checkpointing based rollback recovery techniques can be classified into three categories: *uncoordinated checkpointing, coordinated checkpointing and communication-induced checkpointing*

1.4.1 Uncoordinated Checkpointing

In this kind of checkpointing, each of the process has its own autonomy in deciding when to take checkpoints. This reduces much of the overhead due to communication between processes. The main advantage is lower runtime overhead during normal communication. It also allows process to select its appropriate checkpointing position. However, it also has many shortcomings. First, there is a good chance of domino effect during recovery which can cause the loss of large amount of work. Second, recovery from failure would be slow as it has to find consistent set of checkpoints. Third, it forces each process to have multiple checkpoints and to periodically invoke a garbage collection algorithm to discard the checkpoints which are no longer needed. Fourth, it is not suitable for for applications with frequent output commits as these require global coordination to compute the recovery line.

1.4.2 Coordinated Checkpointing

In this kind of checkpointing, processes orchestrate their checkpointing activities so that checkpoints taken by them will form consistent global state. It simplifies the recovery process as every process will recover from the most recent checkpoint and therefore it is not susceptible to domino effect. One of the major advantage of this method of checkpointing is that it only stores its recent checkpoint on the stable storage which reduces much of the storage overhead cost and eliminates the need for garbage collection. The major disadvantage is that large latency is involved in committing output, as a global checkpoint is needed before an output message is sent. Also, delays and overhead are involved everytime a new global checkpoint is taken.

If perfectly synchronize clocks were available at all the processes, then we would simply ask all the processes to take checkpoint at specific time and it would form a global consistence checkpoints. Since, perfectly synchronize clocks is not available, so the following methods have been proposed for checkpoint consistency; either the sending of the message is blocked for the duration of the protocol or checkpoint indices are piggybacked to avoid blocking.

Blocking coordinated checkpointing

A straightforward approach is to block the processes during the checkpointing interval. This is done by 2-phase protocol. In the first phase, initiator takes local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpoint activity is complete. The coordinator takes a checkpoint and broadcast a request messages to all the processes asking them to take their checkpoint. After receving request message, process stops its execution, flushes all the communication channels, takes a tentative checkpoint and sends an acknowledgement back to the coordinator. After coordinator receives acknowledgement from all the processes, it broadcasts commit message which completes the second phase of the protocol. After receiving commit message, a process removes all permanent checkpoint, makes their tentative checkpoint permanent and resumes its computation. The biggest advantage in this algorithm is that process remains blocked during the checkpointing interval. Therefore, non-blocking schemes are preferable.

Non-Bocking checkpoint coordination

In this approach, processes need not stop their execution while taking checkpoints. A fundamental problem in coordinated checkpointing is to prevent process to receive messages which would make the system inconsistent. If channels are FIFO, this problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, forcing each process to take a checkpoint before receiving the first post-checkpoint message. An example of non-blocking checkpoint coordination protocol using this idea is the snapshot algorithm of Chandy and Lamport in which markers plays the role of checkpointing request messages. In this algorithm, the initiator takes a checkpoint and sends a marker on all outgoing channels. Each process will take a checkpoint when it will receive its first marker and then it will send the marker on all the outgoing channels before sending any application messages. This protocol works assuming the channel are reliable and FIFO.

If the channels are non-FIFO, the following two approaches can be used: first, tha markers can be piggybacked on every post-checkpoint message. When a process receives an application message with a marker, it treats it as if it has received a marker message, followed by the application message. Alternatively, checkpoint indices can server the same role as markers, where a checkpoint is triggered when the receiver's local checkpoint index is lower than the piggybacked checkpoint index.

1.4.3 Communication-induced checkpointing

It is an another way to avoid the domino effect, while allowing checkpoint to take some of their checkpoints independently. Processes maybe forced to take additional checkpoints, and thus process independence is constrained to guarantee the eventual progress of the recovery line. It reduces or completely eliminated the useless checkpoints. It takes two kind of checkpoints, local and forced checkpoints. The checkpoint that the process take independently is called local checkpoints while those that a process is forced to take are forced checkpoints. It piggybacks information in each of the application message. The receive of this application message use this information to determine if it has to take a forced checkpoint to advance the global recovery line. This kind of forced checkpoint is taken before processing the message which introduces the overhead and latency. It is therefore desirable to reduce number of forced checkpoints.

In the further section, we will discuss about some algorithms of coordinated checkpointing.

Much of the previous work [1,4,5] in coordinated checkpointing was focussed mainly on minimizing the number of synchronization message and the number of checkpoints. However all of this algorithms usually blocks the processes during the checkpointing process. Checkpointing includes the time to trace the dependency tree and to save the states of processes on the stable storage, which maybe long. Therefore, blocking algorithms may degrade the performance. To address this issue non-blocking coordinated algorithms [7,8] have been proposed which is discussed in details in the coming sections.

Chapter 2

Koo and Toueg algorithm (IEEE TSE 1987)

2.1 Overview

A checkpoint is basically a saved state to which a process can revert back after a failure and restart from it. There are two ways of checkpoint creation:
i. Checkpoints are taken independently by processes and saved in a stable storage. Upon failure we find a consistent set of checkpoints and revert back.
ii. Coordinated checkpointing where each process saves only its most recent checkpoint and the recent set of checkpoints is always guaranteed to be consistent.
The main problem of the first approach is "domino effect". For applications where time is important, this can cause unacceptable delays. Also, we require large amount of storage in the first approach.
This paper uses the second approach. It also ensures that when a process takes a checkpoint, a minimal number of other processes are forced to take checkpoints.

2.2 Model

1. No memory or clocks are shared by the processes.
2. Messages are sent via FIFO channels.
3. Transmission time is non zero but variable.
4. Failure of processes don't affect the communication network.

2.3 Algorithm

2.3.1 Intuition

There are two kinds of checkpoints : permanent and tentative. We can undo the tentative checkpoint but the permanent checkpoint cannot be undone. The permanent checkpoint always ensures consistency.
Whenever a process wants to create a permanent checkpoint, this algorithm is invoked by it. This algorithm works in two phases:

1. The initiator process p takes a tentative checkpoint and sends out a message to all processes to take tentative checkpoints. If all other processes reply a "yes", then this initiator process p decides that all the tentative checkpoints should be made permanent.

2. Process p's decision is sent to all the processes.

Since all or none of the processes take checkpoints, the recent set of checkpoints is always consistent.

When a process p receives a request to checkpoint from q, it takes a tentative checkpoint only if:
i. there is a received message from p in q's tentative records.
ii. p's latest permanent checkpoint doesn't record sending of that message.

We can clearly see that p has to take a checkpoint or else *p's* recent permanent checkpoint and *q's* latest tentative checkpoint will be inconsistent. So, we have to take a fresh checkpoint at p.

2.3.2 Data Structures

1. $m.l$: Every message m contains a label $m.l$. The value of this label is incremented every time a message is sent. The smallest valued label is $\perp$ and the largest valued label is $\top$.

2. $last\_rmsg_q(p)$: It is the last message q received from p after q took its last permanent or tentative checkpoint.

$$last\_rmsg_q(p) = \begin{cases} m.l, & \text{if m exists} \\ \perp, & \text{otherwise} \end{cases}$$

3. $first\_smsg_q(p)$: It is the first message q sent to p after q took its last permanent or tentative checkpoint.

$$first\_smsg_q(p) = \begin{cases} m.l, & \text{if m exists} \\ \perp, & \text{otherwise} \end{cases}$$

4. $ckpt\_cohort_q$: If q has taken a tentative checkpoint and $last\_rmsg_q(p) > \perp$ before the checkpoint was taken, then p is the *ckpt_cohort* of q. The set of all *ckpt_cohort* of q is represented by $ckpt\_cohort_q$

5. $willing\_to\_ckpt_p$: It is a boolean variable that shows whether process p can take a checkpoint.

2.3.3 Pseudo-code

Daemon Process:

```
SEND(initiator, take a tentative checkpoint and ⊤);
AWAIT(initiator, willing_to_ckpt_initiator);
if willing_to_ckpt_initiator = "yes" then
    SEND(initiator, make tentative checkpoint permanent);
else
    SEND(initiator, undo tentative checkpoint);
```

All processes p:

```
INITIAL STATE
first_smsg_p(daemon) = ⊤;
if p is willing to take checkpoint:
    willing_to_ckpt = "yes"
else:
    willing_to_ckpt = "no"
```

Upon receipt of "take tentative checkpoint and $last\_rmsg_q(p)$" from q:

```
if willing_to_ckpt_p and last_rmsg_q(p) ≥ first_smsg_p(q) > ⊥ then
    take tentative checkpoint
    ∀ r ∈ ckpt_cohort_p SEND(r, take a tentative checkpoint and last_rmsg_p(r));
    ∀ r ∈ ckpt_cohort_p AWAIT(r, willing_to_ckpt_r);
    if ∃ r ∈ ckpt_cohort_p and willing_to_ckpt_r = "no" then
        willing_to_ckpt_p = "no"
SEND(q, willing_to_ckpt_p);
```

Upon receipt of "make tentative checkpoint permanent" or m = "undo tentative checkpoint":

```
if m = "make tentative checkpoint permanent" then
    make tentative checkpoint permanent
else
    undo tentative checkpoint
∀ r ∈ ckpt_cohort_p, SEND(r,m);
```

2.3.4 Explanation

The initiator or the Daemon process starts the algorithm by taking a tentative checkpoint and sending the SEND message asking all other processes to take tentative checkpoints. It also sends $last\_rmsg_q(p)$ to all $p \epsilon\ ckpt\_cohort_q$.

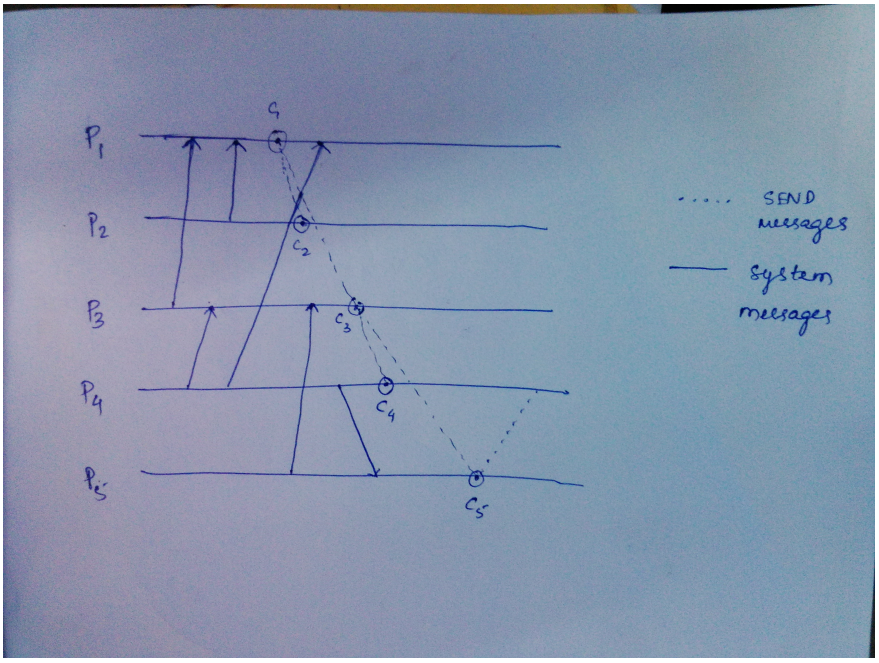
In AWAIT message, the initiator waits for reply from all the processes to which it has sent the SEND message. If it gets a "yes" from all those process, then it sets its *willing_to_checkpoint* to "yes" and sends the message to all the processes to make their tentative checkpoints permanent.

When a process p receives a "take tentative checkpoint" request from q then it checks if $last\_rmsg_q(p) \geq first_smsg_p(q) > \perp$. If this is true and $willing\_to\_ckpt_p$ is true, then p is said to inherit the request and it sends a "take tentative checkpoint" to all process in $ckpt\_cohort_p$. The process p then waits for replies from all its $ckpt\_cohorts$. If all are "yes", then it sets its *willing_to_ckpt* to "yes" and sends its reply to the process q from which it received the request to take checkpoint.

When a process receives the message "make tentative checkpoint permanent" or "undo tentative checkpoint", it takes action accordingly and propagates this to all its $ckpt\_cohorts$. From the time the process p takes a tentative checkpoint to the time it gets a reply from the initiator, the process p doesn't send any system messages.

We can clearly see that a process cannot inherit more than one request to take a tentative checkpoint. This is because from the time a process takes a tentative checkpoint to the time it receives a decision from the initiator, it doesn't send any system message. So $first\_smsg_p(q) = \perp$ for all q and hence p cannot inherit any additional requests.

2.3.5 Example



In this example, process P_2 and process P_3 are the *ckpt_cohorts* of P_1. Note that P_4 is not a *ckpt_cohort* of P_1 because P_1 received the message of P_4 after P_1 took the checkpoint. P_4 and P_5 are the *ckpt_cohorts* of P_3. P_4 is the *ckpt_cohort* of P_5.
P_1 is the initiator process and it first takes a tentative checkpoint and sends "take tentative checkpoint" message to P_2 and P_3.
P_2 takes a tentative checkpoint and replies "yes" to P_1 while P_3 inherits this request and sends "take tentative checkpoint" message to P_4 and P_5.
P_4 takes a tentative checkpoint and replies "yes" to P_3. P_5 takes a tentative checkpoint and sends "take tentative checkpoint" to P_4. Since P_4 has already taken a tentative checkpoint, it replies "yes" to P_5.
Now P_5 sends a "yes" to P_3 and finally P_3 sends a "yes" to the initiator i.e P_1.
Since P_1 gets a "yes" from all its *ckpt_cohorts*, it decides to make the tentative checkpoints permanent and sends the message "make tentative checkpoint permanent" to its cohorts and this is propagated in the same way tentative checkpoints were created.
Finally, we get C_1, C_2, C_3, C_4 and C_5 as the set of permanent checkpoints.

Chapter 3

Cao and Singhal algorithm (IEEE TPDS 1998)

3.1 Overview

This paper presents an efficient coordinated checkpointing algorithm. Two main approaches to reduce the overhead of checkpointing in coordinated algorithms: i) to minimize the number of synchronization messages and the number of checkpoints , ii) to make the checkpointing process nonblocking. Also, they prove the result "There does not exist a nonblocking algorithm that forces only a minimum number of processes to take their checkpoints." Based on this result, this paper proposes an efficient algorithm that neither forces all processes to take checkpoints nor blocks the underlying computation during checkpointing. It also discusses about future research directions in designing coordinated checkpointing algorithms for distributed computing systems.

3.2 Model

There are N processes $P_1, P_2, ..., P_N$. They don't share memory. Mode of communication is message passing. Channels are reliable. Its Asynchronous model, delay during message transmission is finite but arbitrary. Each process knows N or at least upper bound on N. Each process assigns a sequence number to each local checkpoint taken at that process. Each process begins with checkpoint having sequence number 0 and at the end take a checkpoint. The i^{th} checkpoint interval at any process is denoted by all computation performed from checkpoint having sequence number i (including) till checkpoint having sequence number $(i+1)$ (excluding). The messages generated by the

underlying distributed application are referred to as *computation messages*. The messages generated by processes to advance checkpoints are referred to as *system messages*.

Basic Definitions

z-dependency

A process P_q is z-dependent on process P_p during P_ps i^{th} checkpoint interval and during P_qs j^{th} checkpoint interval if P_p sends a message during i^{th} checkpoint interval and P_q receives that message during j^{th} checkpoint interval. It is denoted by $P_p \prec_j^i P_q$. Similarly, transitive z-dependency is denoted by symbol $\prec^*$.

Dependence Record R

Each process P_i maintains a dependence record R_i. Its an array of N bits, one for each process. $R_i[j] = 1$ indicates that process P_i depends on process P_j. Initially, $R_i[i] = 1$ and $R_i[j] = 0$ for $j \neq i$. Whenever P_i sends a computation message to P_j, it appends R_i to message m. Upon receiving m, P_j updates R_j as, $R_j = R_j \cup m.R$ ($\cup$ denotes bitwise OR operation). So, each process stores information about the processes it directly or transitively depends on. Intuition behind this is, if a process takes a checkpoint, all processes that have sent message to that process or there exists a chain of processes in terms of message sending to that process, should be informed about the checkpoint. Other processes need not be informed, since even if they don't take checkpoint, it will not cause inconsistency.

Checkpoint Sequence Number *csn*

Many nonblocking checkpointing algorithms use *csn* as local counter. *csn* is appended to each sent message. Whenever a message m with higher *csn* is received by process P_i, a checkpoint is taken and csn_i is updated to $m.csn$. But this approach requires all processes to take checkpoint for a particular checkpointing initiation so that their *csn*s stay updated. Otherwise, if some process doesn't take checkpoint and update *csn*, when it will send checkpoint requests to other processes, they may not take checkpoints since message *csn* is lower. So, this approach will not work if we want to minimize the number of processes to take checkpoints.

To solve this, each process P_i stores array csn_i of N *csn*s, one for each process. This can be thought as expected *csn* for each process.

3.3 Algorithm

3.3.1 Intuition

A simple nonblocking scheme for checkpointing is as follows. When a process P_i sends a computation message, it piggybacks the current value of $csn_i[i]$. When a process P_j receives a message m from P_i, P_j processes the message if $m.csn \leq csn_j[i]$; otherwise, P_j takes a checkpoint (called forced checkpoint), updates csn_j as $csn_j[i] = m.csn$, and then processes the message. This method may result in a large number of checkpoints. Moreover, it may lead to an avalanche effect, in which processes in the system recursively ask others to take checkpoints.

We can reduce the number of checkpoints taken by observing that, if a process has not sent any message in current checkpointing interval, it doesn't need to take take checkpoint even if *csn* of received message is greater than corresponding entry in *csn* array, since there is no possibility of orphan message causing inconsistency. But this modification is not sufficient to avoid avalanche effect.

When P_j receives message m from P_i and if $m.csn > csn_j[i]$, P_j should take checkpoint if P_i transitively z-depends on P_j in their current checkpoint intervals since by the definition of z-dependency, there is a chain of sent messages from P_j to P_i in current checkpoint intervals. If receiver of a message takes a checkpoint, sender must take a checkpoint. But P_j or m don't have enough information to verify it. So, a checkpoint must be stored to avoid inconsistency. This checkpoint is called 'Forced Checkpoint'. Now, there are 3 types of checkpoints: $forced$, $tentative$ and $permanent$. Tentative and Permanent checkpoints are stored in stable storage. Forced checkpoints can be stored in local memory, even in main memory. When a process takes a tentative checkpoint, it forces all dependent processes to take checkpoints. However, a process taking a forced checkpoint does not require its dependent processes to take checkpoints. Some forced checkpoints may not be necessary. If P_j gets checkpoint request corresponding to initiator P_i, it converts that forced checkpoint into tentative checkpoint. For unnecessary checkpoints, no request will be received since there is no z-dependency. Those will be discarded when tentative checkpoint is made permanent. Requests are not sent in case forced checkpoint is taken. So, avalanche effect is avoided.

3.3.2 Data Structures

Each process P_i stores following variables:

- R_i: Dependence record as described before. It is array of N bits.

- csn_i: Checkpoint sequence number array of N integers.
- $weight$: Real number from $0.0 - 1.0$.
- $first_i$: An array of N bits. $first_i[k] = 1$ indicates that P_i has sent a message to P_k in current checkpoint interval.
- $trigger_i$: A tuple (PID,sn). PID is process id of initiator which caused P_i to take the most recent checkpoint. sn is csn at process P_{PID} when it initiated the checkpointing.
- CP_i: A list of forced checkpoint records. Each record consists of following fields:
  - $checkpoint$: forced checkpoint information.
  - R: Dependence record of P_i from the latest checkpoint (forced or permanent) till current checkpoint.
  - $first$: Array 'first' of P_i from the latest checkpoint (forced or permanent) till current checkpoint.
  - $triggerset$: Set of triggers associated with this forced checkpoint.

3.3.3 Pseudo-code

At process P_i:

```
struct trigger {
  int pid, sn;
};
struct record {
  checkpoint;
  bit R[N];
  bit first[N];
  set<trigger> triggerset;
};

trigger  own_trigger, msg_trigger;
int csn[n];
float weight;
set<record> CP;
set<int> process_set;
bit R[N], temp1[N], temp2[N], first[N];

struct Message {
  int sender, csn;
  bit R[N];
```

```
  trigger msg_trigger;
  message M;
  float weight;
};

// To send computation message to process j:
SEND(msg,j) {
  msg.sender = i;
  msg.R = R;
  msg.csn = csn[i];
  if(first[j]=0):
    first[j]=1;
    msg.trigger = own_trigger;
  else:
    msg.trigger = NULL;
  send(msg,j);
}

// To initiate checkpointing:
INIT() {
  Take local checkpoint on stable storage;
  csn[i]++;
  own_trigger.PID=i; own_trigger.sn=csn[i];
  weight=1.0;
  Message request_msg;
  request_msg.sender=i;
  request_msg.R=R;
  request_msg.csn=csn[i];
  request_msg.trigger=own_trigger;
  for each process j:
    if(R[j]==1):
      weight=weight/2;
      request_msg.weight=weight;
      send(request_msg,j);
}

// Receive computation message:
RECV_COMP_MSG(msg) {
  int j=msg.sender;
  if(msg.csn > csn[j]):
    csn[j] = msg.csn;
    if(msg.trigger != own_trigger):
      if(for all j, first[j]=0 and CP is not empty):
        append msg.trigger to CP[last].triggerset;
      if(for some j, first[j]=1):
        Take a forced checkpoint Record;
        Record.triggerset = {msg.trigger};
        Record.R = R; Record.first = first;
        Reset R; Reset first;
```

```
        append Record to CP;
        own_trigger = msg.trigger;
        csn[i]++;
  process the message;
}

// Receive checkpoint request message:
RECV_REQ_MSG(msg) {
  int j=msg.sender;
  csn[j]=msg.csn;
  if(msg.trigger=own_trigger and
        (for no k, CP[k].triggerset contains msg.trigger) ):
    Send reply_msg to own_trigger.PID with same weight as received;
    exit();
  if(msg.trigger!=own_trigger and (for all j, first[j]=0) ):
    Send reply_msg to own_trigger.PID with same weight as received;
    exit();
  if(msg.trigger=own_trigger):
    find k such that CP[k].triggerset contains msg.trigger;
    save CP[k].checkpoint on stable storage;
    temp1=CP[k].R;
    for y=0 to k-1:
      temp1= temp1 OR CP[y].R;
  else:
    take local checkpoint on stable storage;
    csn[i]++;
    own_trigger=msg.trigger;
    temp1=R;
    for each record in CP:
      temp1= temp1 OR record.R;
  temp2=temp1 EXCLUDE msg.R; // x EXCLUDE y=1 iff x=1 and y=0
  temp1=temp1 OR msg.R;
  weight = msg.weight;
  Message request_msg;
  request_msg.sender=i;
  request_msg.R=temp1;
  request_msg.csn=csn[i];
  request_msg.trigger=msg.trigger;
  for each process j:
    if(temp2[j]==1):
      weight=weight/2;
      request_msg.weight=weight;
      send(request_msg,j);
  Send reply_msg to own_trigger.PID with remaining weight;
  Reset R; Reset first; // tentative checkpoint has been taken
}

// Receive Reply message:
RECV_REP_MSG(msg) {
```

```
  int j=msg.sender;
  insert j in process_set;
  weight=weight+msg.weight;
  if(weight=1.0):
    for each process k in process_set:
      commit_msg msg;
      msg.trigger=own_trigger;
      send(msg,k);
    clear process_set;
}

// Receive commit message:
RECV_COMMIT_MSG(msg) {
  make tentative checkpoint permanent;
  if there exists k, CP[k].triggerset contains msg.trigger:
    for y=k+1 to N:
      R = R OR CP[y].R;
      first = first OR CP[y].first;
  clear CP;
}
```

3.3.4 Explanation

This algorithm assumes that at a single time, only one process initiates checkpointing process. It can be modified for multiple initiations.

To initiate checkpointing, process P_i sets its weight to 1. P_i sends request message to each process P_j on which it depends on i.e. $R_i[j] = 1$. While each sending, weight is divided by 2, so half weight is sent and half remaining. Each request message contains csn, R_i, initiator trigger and weight.

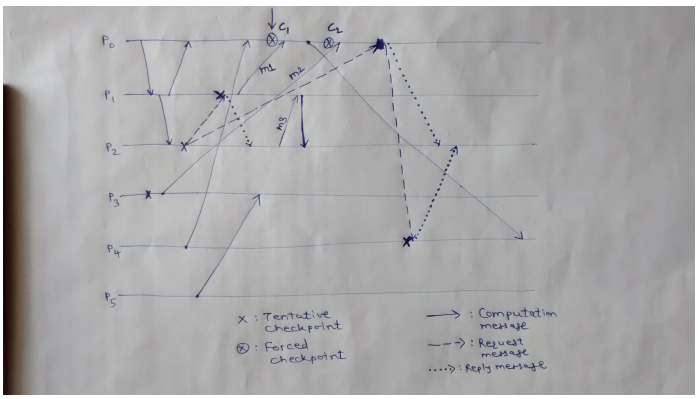
When P_i receives a computation message from P_j, if $msg.csn \leq csn_i[j]$, P_i just processes message and exits. Otherwise, update $csn_i[j] = msg.csn$. If $msg.trigger = owntrigger$, it indicates that P_i has taken checkpoint corresponding to same initiator. So, no new checkpoint is taken. Just process message and exit. Otherwise, if P_i has not sent any message in this checkpointing interval, it does not need to take new forced checkpoint. It just appends msg.trigger to latest forced checkpoint triggerset. If it has sent a message, it takes forced checkpoint with msg.trigger in triggerset and updates $P_i.trigger = msg.trigger$. This checkpoint information contains R_i and $first_i$ which were for that checkpoint interval. R_i and $first_i$ are reset for new checkpoint interval. $csn_i[i]$ is incremented. Then P_i processes message and exits.

When P_i receives a request message msg from P_j, if P_i trigger equals msg.trigger and there is no forced checkpoint for this trigger, it implies that P_i has not taken a checkpoint after getting computation message from another

process with same trigger according to the rules stated in previous paragraph. So, it does not need to store any forced checkpoint to stable storage. It just replies initiator with received weight. If P_i has taken a forced checkpoint, it stores that checkpoint to stable storage as tentative checkpoint. If msg.trigger is different and P_i has not sent any message in current checkpoint interval, it doesn't need to take checkpoint, just reply to initiator with received weight. If P_i has sent a message, it takes a tentative checkpoint. Whenever P_i takes tentative checkpoint, it computes dependence record R corresponding to that checkpoint. P_i sends request message to those processes that P_i depends on but to whom no request has been sent ($R_i = 1$ and $msg.R = 0$). weight computation is done as in case of initiator, starting with received weight. Reply is sent to initiator with remaining weight.

Termination and garbage collection: The checkpoint initiator adds weights received in all reply messages to its own weight. When its weight becomes equal to one, it concludes that all processes involved in the checkpointing process have taken their tentative local checkpoints. Then it sends out commit messages to all processes from which it received replies. Processes make their tentative checkpoints permanent on receiving the commit messages. The older permanent local checkpoints and all forced checkpoints at these processes are discarded because a process will never roll back to a point prior to the newly committed checkpoint. When a process discards its forced checkpoints, it also updates its data structures CP, R and first by taking union of arrays during consecutive checkpoint intervals.

3.3.5 Example



In above figure, P_3 takes a tentative checkpoint. Since it has no dependency on other processes, it doesn't send any request message. P_2 takes a tentative checkpoint and sends requests to P_1 and P_0 since it has received messages from these two processes.

When P_1 receives request, even if it has received a message from P_0 in current checkpoint interval, it doesn't send a request to P_0 since R in request message indicates that a request has already been sent to P_0. P_1 takes tentative checkpoint and sends a reply to initiator P_2. When P_0 receives $m1$ from P_1, it contains trigger corresponding to P_2. Since this trigger differs from own trigger and P_0 has sent a message in current checkpoint interval, it takes a forced checkpoint C_1. For forced checkpoint, no request is sent. When P_0 receives $m2$ from P_3, it takes another forced checkpoint C_2. After that, P_0 receives request sent by P_2. Now, it makes corresponding forced checkpoint C_1 tentative after comparing triggers of request and C_1. In C_1 checkpoint interval, P_0 had received message from P_4 and P_1. It only sends request to P_4 by checking R in request message and sends reply with remaining weight to initiator P_2. P_4 receives request and sends reply to P_2. When P_2 gets all reply messages, its weight adds up to 1, it sends commit messages to P_0, P_1 and P_4. Forced checkpoint C_2 on P_0 will be discarded. P_2, P_0, P_1 and P_4 make their tentative checkpoints permanent. In this way, this nonblocking algorithm does not force all processes to take checkpoints and works efficiently.

If we compare with basic algorithm, when P_0 takes checkpoint C_2, it forces P_1 also to take checkpoint, since it has sent a message and received a message from P_1 in current checkpoint interval. Then P_1 will also force P_2 to take checkpoint due to $m3$. So, many unnecessary checkpoints are taken. This sequence may also continue indefinitely and lead to avalanche effect. Present algorithm avoids this by not forcing other processes after taking forced checkpoints. Unnecessary checkpoints are discarded.

Chapter 4

Cao and Singhal mutable checkpoint algorithm (IEEE TPDS 2001)

4.1 Overview

In our previous algo, we proved that there doesn't exist any non-blocking algorithm which forces only minimum number of processes to take their checkpoints. So, here we will describe a non-blocking algorithm which relaxes the min-process condition but still tries to minimize the number of checkpoints saved on the stable storage. The proposed algorithm will use the concept of "mutable checkpoint" which is neither a tentative checkpoint nor the permanent checkpoint. This mutable checkpoints can be saved anywhere i.e., the main memory or the local disk. This way it reduces much of the overhead of transferring data to stable storage at the file server across the network. It also tries to avoid the avalanche effect.

4.2 Model

4.2.1 Computation Model

The description of this model is same as before except that it don't require to know the upper bound on N. So in this model of computation, we are assuming it to be asynchronous and reliable. Moreover, the message generated by the underlying distributed application will be referred to as *computation messages* whereas the messages generated by the process to advance checkpoints will be referred to as *system messages.*

4.2.2 Basic ideas behind non-blocking algorithms

Most existing coordinated checkpointing algorithms use 2-phase protocol. In the first phase, initiator ask all the relevant processes to take their tentative checkpoint and then the processes replies to the initiator. When initiator gets reply from all the processes then only it enters the second phase. In between these, it asks the processes to remains blocked.

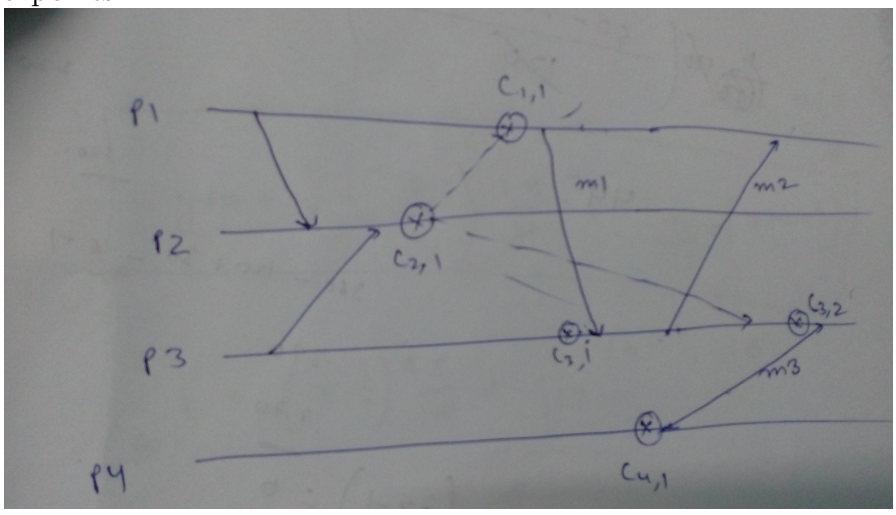
In case of non-blocking algorithms, initiator does not require any process to get blocked, that means all relevant processes can still get the computation messages from others which need to be dealt with to avoid inconsistency. This can be done by using the concept of sequence number as described in the previous algorithm.

4.3 Algorithm

We are assuming that at most one checkpointing will be in progress in the system.

4.3.1 Intuition

The above method (sequence number concept) of using checkpointing may result in large number of checkpoints. Moreover, this may lead to avalanche effect, in which the processes in the system recursively ask other to take checkpoints.



For example, in the above figure, P_2 takes its own checkpoint and send the checkpointing request to P_1 and P_3. When the request reaches P_1, P_1 takes the checkpoint and then send the message m1 to P_3. When message

m1 reaches P_3, it has to take a checkpoint before processsing message as m1 csn $>csn_3[1]$. P_4 has not communicated with with other processes before it takes a local checkpoint. Later is send m3 to P_3, and before processing this message, P_3 has to take a checkpoint as m3 csn $>csn_3[4]$. Then P_3 requires P_1 to take another checkpoint (not shown in the figure) due to m1. If P_1 had received message from other processes after it sent m1, then those processes would have been forced to take checkpoints. This chain may never end.

Now we will describe two observations from above figure which will help us to reduce the number of checkpoints.

Observation 1: Checkpoint $C_{3,2}$ is not necessary even though m2 exist, since P_3 will not receive checkpoint request associated with $C_{4,1}$. we can also observe that m2 will not become orphan even if we don't take $C_{3,2}$.

Observation 2: P_4 does not have enough information to know if it will receive a checkpoint request associated with $C_{4,1}$ when P_3 receives m3.

These observation imply that $C_{3,2}$ is unnecessary but still unavoidable. Thus there are two kinds of checkpoint correspoding to computation messages. One is of the form $C_{3,1}$ which is associated with initiator P_2 and will receive checkpoint request after sometime. Other one is $C_{3,2}$ which is associated with initiator P_4 and will not receive checkpoint request in future.

We introduce a new concept of *mutable checkpoint*, to reflect the difference between these kind of checkpoints ($C_{3,1}$,$C_{3,2}$). A mutable checkpoint is neither a tentative checkpoint nor a permanent checkpoint, but it can be turned into tentative checkpoint. In this case, whenever a process takes mutable checkpoint, it will not send request to all the relevant processes to take checkpoints and it does not need to save this mutable checkpoint into stable storage. It can save mutable checkpoint anywhere, i.e., either in the main memory or the local disk. In this case, whenever P_i takes a mutable checkpoint, it will convert it into tentative checkpoint whenever it gets checkpointing request. If the process does not receive a checkpointing request, it discard the mutable checkpoint after the checkpointing activitiy terminates.

In the above figure, when m1 arrives at P_3, it takes $C_{3,1}$ as a mutable checkpoint and change it into tentative checkpoint when it receives checkpointing request from P_2. Whereas when P_3 receives m3, it does not need to take mutable checkpoint if checkpointing activity associated with $C_{4,1}$ finished. Otherwise, it will take the mutable checkpoint which will be discarded later when P_4 checkpoint terminates.

In the above described alogrithm, a process can still take unnecessary checkpoints but this can be avoided by the following method. When a process P_i sends a checkpoint request to P_j, it attaches csn_i[j] to the request. On receiving the request, P_j compares the attached csn_i[j] with its own csn_j[j]. If csn_j[j] $>csn_i$[j], P_j does not need a checkpoint otherwise, takes a checkpoint.

4.3.2 Data Structure

Each of the process in our proposed algorithm will store the following:
R_i, csn_i, weight, $trigger_i$: defined in the above algorithm
$sent_i$: its a boolean variable which would be set to 1, if the process P_i has sent any message in current checkpoint interval.
cp_state_i: its a boolean variable which would be set to 1, if process P_i is involved in the current checkpoint process.
old_csn: a variable which stores the value of current tentative (permanent) checkpoint.
CP_i: It's like a record maintained by each of the process and consist of the following field:

mutable: the mutable checkpoint of P_i

R: It's own boolean vector before it takes the current mutable checkpoint.

trigger: the trigger which is associated with the current mutable checkpoint.

sent: It's own sent before it takes current mutable checkpoint.

csn will be intialized to an array of 0's at all processes.
The trigger tuple at process P_i is initialized to (i; 0).
The weight and cp_state at a process is initialized to 0.

4.3.3 Pseudo Code

Pseudo Code for P_i when it sends a computation message to P_j:

```
if(cp_state == 1):
    send(P, message, csn[i], owntrigger);
    sent = 1;
else:
    send(P, message, csn[i], NULL);
    sent = 1;
```

Actions for initiator P_j:

```
// Initialization of variable
csn[j]++;
owntrigger = (P,csn[j]);
cp_state = 1;

for(k = 0 to N)
    MR[k]csn = 0
    MR[k]R = 0
MR[j]csn = csn[j]
MR[j]R = 1
prop_cp(R,MR,P,owntrigger,1.0); // propagate checkpoint request
```

```
take a local checkpoint (on stable storage)
old_csn = csn[j]
sent = 0
resent R
```

Action for process P_i on receiving checkpoint request from P_j:

```
receive(recv_P,request,MR,recv_csn,msg_trigger,req_csn,recv_weight):
    csn[j] = recv_csn;
    if old_csn > req_csn:
        send(P,reply,recv_weight) to the initiator
        return
    cp_state = 1
    if(msg_trigger == own_trigger):
        if(CPtrigger == msg_trigger):
            prop_cp(CP_R,MR,P,msg_trigger,recv_weight)
            save CP mutable on stable storage
            old_csn = csn[i]
            CP = NULL
            send(P,reply,weight) to the initiator
        else:
            send(P,reply,recv_weight) to the initiator
    else:
        csn[i]++
        own_trigger = msg_trigger
        prop_cp(R,MR,P,msg_trigger,recv_weight)
        take a local checkpoint on the stable storage
        old_csn = csn[i]
        send(P,reply,weight) to the initiator
        sent = 0
        Reset R
```

Action at process P_i on receiving a computation message from P_j:

```
receive(recv_P, m, recv_csn, msg_trigger):
    if(recv_csn <= csn[j]):
        R[j] = 1
        process the message
    else if(csn[msg_trigger pid] >= msg_trigger inum):
        csn[j] = recv_csn;
        R[j] = 1
        process the message
    else:
        csn[j] = recv_csn
        if(msg_trigger!=Null and sent=1 and
        message_trigger!=own_trigger)
            take a local checkpoint
            save it in CP mutable
            CP_trigger = msg_trigger
            CPR = R
            CPsent = sent
```

```
            sent = 0
            Reset R
        if(msg_trigger != Null and cp_state = 0)
            cp_state = 1
            csn[i]++
            own_trigger = msg_trigger
            R[j] = 1
            process the message
    prop_cp(R,MR,P,msg_trigger,recv_weight)
    weight = recv_weight
    for(k=0 to N):
        temp[k]csn = max(MR[k]csn,csn[k])
        temp[k]R = max(MR[k]R,R[k])
    for any Pk such that R[k] = 1 and
    (max(MR[k]csn,csn[k])!=MR[k]csn)
        weight = weight/2
        send(P,request,temp,csn[i],msg_trigger,csn[k],weight)
```

Actions in the second phase for the initiator P_i:

```
receive(recv_P,reply,recv_weight)
weight = weigth+recv_weight
if weight == 1
    cp_state = 0
    broadcast(commit,msg_trigger)
```

Action at other process P_j on receiving a broadcast message:

```
receive(commit,msg_trigger)
csn[msg_trigger pid] = msg_trigger inum
cp_state = 0
if CP_trigger = msg_trigger and CP != Null
    sent = sent U CPsent
    R = R U CPR
    CP = Null
if there is a tentative checkpoint associated with msg_trigger,
make it permanent.
```

4.3.4 Explanation

To clearly present the algorithm, we assume that at any time, atmost one checkpoint is in progress.

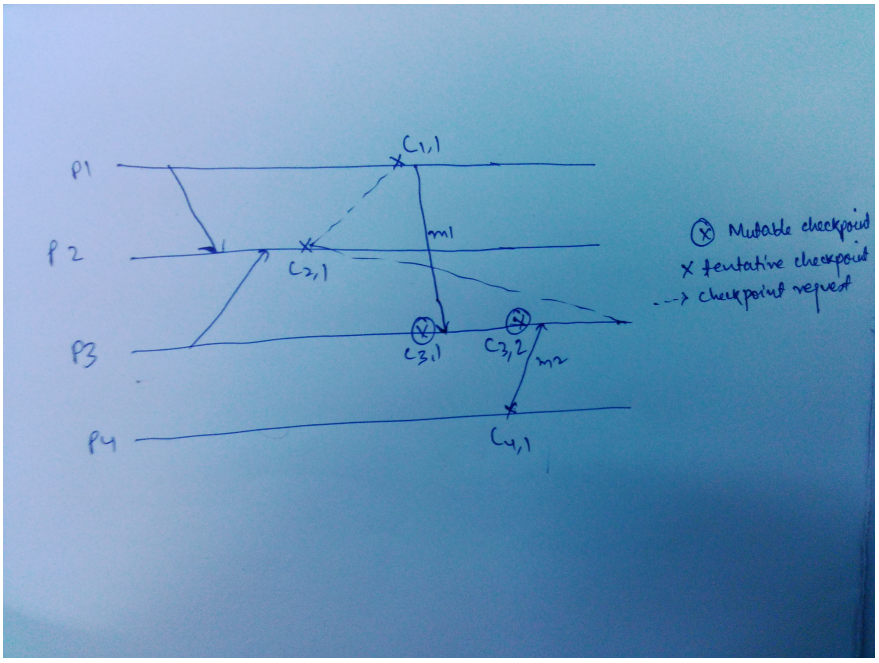
Any process can initiate checkpointing process. When a process P_i initiates checkpointing, it takes a local checkpoint, increment its csn_i[i], sets $weight_i$ to 1, set cp_state_i to 1 and stores its own identifier and the new csn_i[i] in its trigger. Then it sends the checkpointing request to each P_j such that R_i[j] = 1 and resumes its computation.

When a process P_i receives a request from P_j, it will first compares req_csn with its old_csn to check whether it has to inherit that request. If it does not have to inherit the request, it will reply back to the initiator with the appended weight and then exits. Otherwise it updates its local variable and compares msg_trigger with its own_trigger. If both are same, then it checks whether there is a mutable checkpoint having the same trigger as msg_trigger. If not, it sends the appended weight to initiator, otherwise it saves the mutable checkpoint onto the stable storage and then will propagate the request as given below. For each process P_k on which P_i depends but not P_j, P_i sends request to P_k. Also, P_i will append the initiator trigger and portion of received weight into all the requests. At last, it will send reply to the initiator with the weight equal to the remaining weight and resumes its computation. If msg_trigger is not equal to own_trigger, P_i takes a tentative checkpoint, increase its csn_i[i] and propagate the requests as described above. It then resets its local variable and then resumes its computation.

When P_i receives computation message from P_j, P_i compares m csn with its local csn_i[j]. If m csn is less than equal csn_i[j], the message is processed and no checkpoint is taken. Otherwise, it will imply that P_j has taken local checkpoint before sending the message. Process P_i updates csn_i[j] to m csn and checks if the following condition are satisfied.
Condition 1: P_j is in checkpointing process before sending m.
Condition 2: P_i has sent a message since last checkpoint.
Condition 3: P_i has not taken a checkpoint associated with the initiator.
If all of them are satisfied, P_i takes a mutable checkpoint and updates its data structure.

During termination and garbage collection, the initiator adds weight received in all reply messages to its own weight. When its weight become equal to 1, it concludes that all processes involved in the checkpointing have taken their tentative checkpoints. Then it broadcasts commit message to all processes in the system. If any process has taken tentative checkpoint, on receiving commit message, it will change the tentative checkpoint into permanent checkpoint and clears cp_state. Other processes discard their mutable checkpoints if they have any and it also clears its cp_state.

4.3.5 Example



In the above example, we can see that process P_2 starts the checkpointing process and takes tentative checkpoint and then it propagates the checkpointing request to process P_1 and P_3 as $R_2[1]$ and $R_2[3]$ are 1. When process P_1 receives checkpointing request, it takes tentative checkpoint and then sends the message m1 to process P_3. Since m1 csn $> csn_3[1]$ and it sends the message during current checkpoint interval, it will take mutable checkpoint before processing the message. If process P_4 finished checkpointing process before sending message m2, than P_3 does not need to take the checkpoint $C_{3,2}$ else it will take a mutable checkpoint before processing m2. When P_3 receives checkpointing request from P_2, since $C_{3,1}$ is a mutable checkpoint associated with P_2, P_3 turns $C_{3,1}$ into a tentative checkpoint by saving it into a stable storage. Finally the checkpointing initiated by P_2 terminates when checkpoints $C_{1,1}$, $C_{2,1}$ and $C_{3,1}$ are made permanent. P_3 discards $C_{3,2}$ when it makes $C_{3,1}$ permanent or receives P_4 commit, whichever is earlier.

Chapter 5

Conclusion

Most of our previous work on coordinated checkpointing was based on blocking the processes during the checkpointing interval. Here we have described three different non-blocking algorithms.

In the Koo and Toueg algorithm, the system has more tendency to tolerate failure in contrast to the previous algo. During checkpointing, they make sure that minimum number of additional processes are forced to take checkpoint. Similarly, minimum number of additional processes are forced to restart when any of the process get restart.

In Cao and Singhal algorithm, with the help of new concept "z-dependence" we prove a general result, there does not exist a non-blocking algorithm which only forces minimum number of processes to take checkpoint. In our case, forced checkpoint is neither a tentative nor permanent checkpoint but it can be turned into tentative checkpoint. It also avoids avalanche effect and significantly reduces the number of checkpoints.

In Cao and Singhal algorithm using mutable checkpoints, we relaxed some condition and present non-blocking algorithm. More importantly, we proposed the concept of "mutable checkpoint" in implementing the non-blocking algorithm. Mutable checkpoints can be saved anywhere i.e., on local disk or main memory. In this way, taking mutable checkpoints avoid the overhead of transmitting data into stable storage. Based on mutable checkpoints, our non-blocking algorithms avoid the avalanche effect and forces only a minimum number of processes to take checkpoint on the stable storage.

We proposed two types of algorithms: first one is Koo-Toueg which is a blocking algorithm and the other two is non-blocking algorithm. In this section, we will compare the performances of these two types of algorithms.

C_{uni}: cost of sending a message from one process to another process.
C_{broad}: cost of broadcasting a message to all processes.
T_{disk}: delay incurred in saving a checkpoint on the stable storage.

T_{data}: delay incurred in transferring a checkpoint to stable storage.
T_{msg}: delay incurred by system message during checkpointing process.
T_{ch}: checkpointing time. $T_{ch} = T_{msg} + T_{data} + T_{disk}$
N_{min}: Minimum number of process that need to take checkpoint using Koo-Toueg algo.
N_{dep}: Average number of process on which a process depends.

Comparison		
	Koo-Toueg	Non-blocking
Checkpoints	N_{min}	N_{min}
Blocking time	N_{min} * T_{ch}	0
Messages	3*N_{min}*N_{dep}*C_{uni}	2*N_{min}*C_{uni}+min(N_{min}*C_{uni},C_{broad})
Distributed	Yes	Yes

References

[1] R. Koo and S.Toueg "Checkpointing and Rollback Recovery for Distributed Systems", IEEE Trans. Software Eng., vol. 13, no. 1, pp. 2331, Jan. 1987

[2] G. Cao, M. Singhal, On coordinated checkpointing in distributed systems, IEEE Trans. Parallel Distributed System 9 (12) (1998a) 12131225

[3] G. Cao, M. Singhal, Checkpointing with mutalbe checkpoints, IEEE Trans, Parallel Distributed System

[4] Y. Deng, E.K. Park, Checkpointing and rollback-recovery algorithms in distributed systems, J. Systems Software 4 (1994) 59 71.

[5] J.L. Kim, T. Park, An e cient protocol for checkpointing recovery in distributed systems, IEEE Trans. Parallel Distributed Systems 5 (8) (1993) 955960.

[6] E.N. Elnozahy, D.B. Johnson, W. Zwaenepoel, The performance of consistent checkpointing, Proc. 11th Symp. on Reliable Distributed Systems, IEEE Press, New York, 1992, pp. 86 95.

[7] L.M. Silva, J.G. Silva, Global checkpointing for distributed programs, Proc. 11th Symp. on Reliable Distributed Systems, Houston, 1992, pp. 155 162.