Checkpointing and Recovery for Distributed Shared Memory Applications

Jinsong Ouyang and Gernot Heiser
School of Computer Science and Engineering
University of New South Wales
Sydney 2052, Australia
jinsong@cse.unsw.edu.au

Abstract

This paper proposes an approach for adding fault tolerance, based on consistent checkpointing, to distributed shared memory applications. Two different mechanisms are presented to efficiently address the issue of message losses due to either site failures or unreliable non-FIFO channels. Both guarantee a correct and efficient recovery from a consistent distributed system state following a failure. A variant of the two-phase commit protocol is employed such that the communication overhead required to take a consistent checkpoint is the same as that of systems using a one-phase commit protocol, while our protocol utilises stable storage more efficiently. A consistent checkpoint is committed when the first phase of the protocol finishes.

1 Introduction

Checkpointing and rollback recovery are well known mechanisms which can be used to provide fault tolerance in distributed systems, and one of the key issues is to provide an efficient and light-weight mechanism which collects checkpoints of individual processes in the distributed environment to form a *consistent distributed system state*, a system state reachable through some correct execution of the distributed processes.

In this paper, we present a novel approach to provide fault tolerance for distributed shared memory applications. Compared to previous systems, our approach particularly focusses on the following issues:

- reducing the communication overhead required to construct a consistent distributed system state, which is particularly important for software-based systems;
- efficiently addressing the problem of message losses due to either site failures or unreliable non-FIFO channels, to guarantee a correct and efficient recovery following a failure (two different mechanisms are provided).

This paper is organised as follows. In Section 2, we describe the concept of consistency and the two types of mechanisms for constructing a consistent distributed system state. In Section 3, we describe our system model. In Sections 4 and 5, we describe our checkpointing and rollback recovery algorithms. We present the conclusion in Section 6.

2 Background

The key issue of supporting fault tolerance in distributed systems using checkpointing and rollback recovery is how to obtain a consistent state of a distributed system. Chandy and Lamport [5] formally defined the concept of a consistent distributed system state, and introduced an algorithm by which a process in a distributed system determines a global state of the system during a computation.

Briefly, a set of process states forms a consistent distributed system state if it satisfies the following condition: *For each message among the processes, if it is recorded in the state of the receiving process, it must also be recorded in the state of the sending process*. Informally, we can use a time diagram to describe a system's execution, where horizontal lines are time axes of executing processes, and messages are represented by arrows. For example, in Figure 1, P_1, P_2, P_3, and P_4 are four processes, and a, b, and c are cuts (sets of process states) each of which forms a distributed system state.

According to the definition, cuts b and c are consistent cuts, while cut a is an inconsistent cut, as process P_1 recorded its state after it received the message while process P_2 recorded its state before it sent the message. If the system restarts from system state a, process P_1 restarts from a point where it already received the message from P_2, but P_2 restarts from a point where it has not sent the message to P_1, so process P_1 will actually receive the message from P_2 twice. This incorrect execution results from the inconsistency of cut a. Another important fact is that although cut b is a consistent distributed system state, the messages to processes P_1, P_3, and P_4 must be recorded in some way, oth-

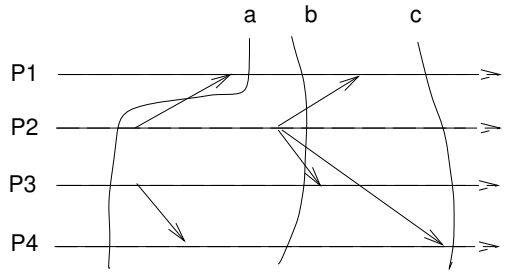

Figure 1: Consistent and inconsistent system states

erwise message losses will occur if the system restarts from state b. For example, when checkpointing state b in Figure 1, the mechanism proposed below ensures that the messages in transit at the time of the checkpoint are logged, ensuring that a consistent distributed system state can be recovered from that checkpoint.

There have been many approaches using checkpointing and/or message logging mechanisms to provide fault tolerance in distributed systems, and their emphasis was generally on how to construct a consistent state from which the system can restart if a failure occurs later. According to when and how a consistent state of a distributed system is built, the existing systems can be divided into two classes as follows:

1. *Independent checkpointing and message logging*: In this type of system [9, 20, 21], the main idea is that processes do not need to synchronise with one another during the checkpointing and message logging phases, which means that individual processes perform their message logging and checkpointing independently, reducing communication overhead in this phase. With message logging, every process can detect its dependency on the states of other processes with which it communicates, and the dependency control information enables a reconstruction of a consistent distributed system state following a failure, using process rollback and message replay. So, this type of approach focusses on reducing communication overhead during the checkpointing and message logging phases, and puts most work into the recovery phase. It is assumed in these systems that failures are infrequent.

2. *Consistent checkpointing*: This type of system [7, 10, 12, 17, 19, 22] attempts to construct a consistent distributed system state in a checkpointing phase. Checkpointing of processes is synchronised in such a way that the resulting set of checkpoints forms a consistent distributed system state; consequently, this makes rollback recovery less expensive.

Compared to consistent checkpointing, independent checkpointing and message logging has a significant message logging overhead and potentially significantly increased memory requirements. This overhead may outweigh the gains through avoiding the synchronisation overhead during the checkpointing phases. Moreover, independent checkpointing and message logging makes recovery expensive.

Our approach is therefore based on *consistent checkpointing*. Compared to previous systems, it has the following features:

- Two different mechanisms are provided to address the issue of message losses due to site failures or unreliable non-FIFO channels. Both ensure that a checkpoint of a consistent state can only be committed if no messages are in transit or lost. This guarantees a correct and efficient recovery from a consistent distributed system state following a failure.

- Consistent checkpoints are taken efficiently. Unlike other systems using a two-phase commit protocol, our approach implements the second phase of consistent checkpointing in a lazy way which does not require any extra message exchange in the system, and does not delay committing a consistent checkpoint until the second phase of consistent checkpointing terminates. While the communication overhead required to take a consistent checkpoint is the same as that of systems using a one-phase commit protocol, our approach utilises stable storage more efficiently. We first take *tentative* checkpoints, which are made *permanent* after the first phase of the two-phase protocol, at which time the previous checkpoints can be discarded, resulting in more efficient use of stable storage. Systems using a one-phase commit protocol must always keep the two most recent checkpoints for each process.

3 System model

3.1 Assumption

Our work is partially motivated by the systems [17, 18, 19], and focusses on the above issues which were not addressed in the previous systems. We make the following assumptions about the distributed environment on which our model is built:

1. nodes fail by stopping. The failed processes can be relocated to some other working node, and the process states can be recovered with the checkpoints stored on stable storage;

2. the network channels are unreliable non-FIFO channels which may lose or reorder messages, and may

be temporarily broken. A reliable message delivery can be realized by retransmitting a message a number of times until an acknowledgement is received from the destination process. If no acknowledgement is received within a timeout interval, an error due to either a node failure or a temporarily partitioned channel is assumed to have occurred;

3. all the processes involved in a consistent checkpoint or a rollback recovery belong to a single distributed application, checkpointing or recovery of different distributed applications does not interfere with each other;

4. for each distributed application, there is one *fault tolerance support manager* (FTSM) on each node responsible for checkpointing and recovery of processes within this application. In our implementation, the FTSM will be a component of the DSM runtime system;

5. processes communicate with each other through distributed shared memory[1].

3.2 Distributed shared memory model

In this section, we briefly describe one of the typical distributed shared memory models [1, 4, 6, 8, 11, 13] on which our system is built—release consistency [4, 8].

In the *release consistency* model, not only is each shared memory access classified either as a synchronisation access or an ordinary access, but synchronisation accesses must be classified as acquire and release accesses. Formally, a system is release consistent if [4]:

1. before any ordinary access is allowed to perform with respect to any other processor, all previous acquires must be performed;

2. before a release is allowed to perform with respect to any other processor, all previous ordinary accesses must be performed;

3. synchronisation accesses must be sequentially consistent with each other.

Release consistency is a consistency model which, compared to stricter consistency models, reduces the number of messages required to maintain consistency in a DSM. Informally, consistency is guaranteed only at specific synchronisation points at which ordinary accesses are pipelined or buffered between synchronisation accesses; this relaxed consistency model results in higher efficiency.

[1]Our model can be built on either message-passing systems or distributed shared memory systems.

4 Distributed checkpointing

4.1 Efficient consistent checkpointing

This section describes in principle how the consistent checkpointing algorithm efficiently constructs a consistent distributed system state. We first describe the techniques used for consistent checkpointing.

- For each distributed application, there is one distinguished FTSM on a node which acts as the coordinator of checkpointing and recovery.

- Like some other systems [7, 17, 19], each consistent checkpoint is uniquely identified by an increasing *checkpointing sequence number* (CSN), and each normal message delivered in the system is tagged with the current CSN of the sender. Besides, each normal message is also tagged with the *status bit* of the sender. This bit is 0 if the current checkpoint of the sender is tentative, otherwise the current checkpoint is permanent.

- A variant of a two-phase commit protocol is employed. This protocol has the communication overhead of a one-phase commit protocol without delaying committing a consistent checkpoint. Furthermore, after the current checkpoint becomes permanent, the previous checkpoint can be deleted to save stable storage space. (Systems using a one-phase commit protocol must always keep the last two checkpoints for each process.) The second phase of checkpointing is implemented in a lazy fashion in that the decision of the coordinator will be delivered to other processes by the status bit (see above) piggybacked on each normal message delivered in the system. If, after the coordinator makes the current checkpoint permanent, there are no more messages sent from the coordinator node to any of the other nodes, each process on the other nodes needs to keep its last two checkpoints, as it does not know the decision of the coordinator. In this worst case scenario, the checkpoints do not become permanent until the next consistent checkpoint is initiated, and the storage overhead is equal to that of a system based on one phase commit.

In principle, when the coordinator initiates a new consistent checkpoint, it takes tentative checkpoints of all local processes belonging to the application, and informs other FTSMs to take tentative checkpoints of their local processes. This is done through *marker* messages containing the current CSN. When a node receives any message whose CSN is bigger than the local one, the local FTSM takes tentative checkpoints of the local processes, increments its local CSN, and replies to the coordinator. The coordinator will set its

status bit once it is informed that all the processes within the application have been checkpointed and there are neither messages in transit nor message losses. (Techniques for achieving this will be described in the next subsection.) Afterwards, if a node receives a message whose status bit is set while the local status bit is 0, the local checkpoints are made permanent, the status bit is set, and the previous checkpoints are discarded.

With the underlying distributed shared memory model, release consistency, we can make the following optimisation to further reduce the consistent checkpointing overhead: A new consistent checkpoint can be triggered by such events as the expiry of a time interval, a certain number of release accesses performed, or an output to the outside world [10]. For example, when the processor on which the coordinator resides is about to perform a release, and the number of releases performed exceeds a predefined number, the coordinator may start a new consistent checkpoint at this time. The coordinator first checkpoints the local processes, and tags any update messages with its CSN so that the nodes to which the update messages will be delivered do not need the extra marker messages. By this optimisation, the marker messages will only be sent to nodes which are not sent any update messages, and the number of messages is further reduced.

Using these mechanisms, our approach can efficiently construct a consistent state of a distributed application with minimum communication overhead and stable storage requirements.

4.2 Dealing with message losses

We have not yet addressed the important issue of message losses due to either site failures or unreliable non-FIFO channels. Message losses due to site failures can occur as follows: a message was sent before the sender takes its checkpoint, whereas it has not been received by the receiver when the receiver fails after taking its checkpoint and replying to the coordinator. Even with a reliable transport protocol (e.g. TCP), a message can be lost during delivery (e.g. due to a temporarily broken channel). If message losses cannot be solved properly, a correct recovery from a consistent distributed system state cannot be guaranteed.

Figure 2 is an example in which message losses occur: P_1, P_2, and P_3 are three processes of a distributed application on three different nodes. Process P_1 is the coordinator of consistent checkpointing and recovery. It starts the ith consistent checkpoint at some point, and sends marker messages to P_2 and P_3. P_2 takes a tentative checkpoint when receiving the marker message from P_1. Before receiving the marker message, P_3 receives a message from P_2 which is sent after P_2 takes its checkpoint, and P_3 takes its tentative checkpoint before changing its local state. The coordinator P_1 receives the replies from both P_2 and P_3 at point j, and makes its checkpoint permanent by setting its status bit. Now, we assume that P_1 fails for some reason. During recovery, it informs all processes to roll back to their ith checkpoints, and restart execution from there. However, this will cause an incorrect recovery because message M from P_3 to P_2 will be lost: P_3 sent M before its ith checkpoint while P_2 received M after its ith checkpoint, and M is not recorded by P_2.

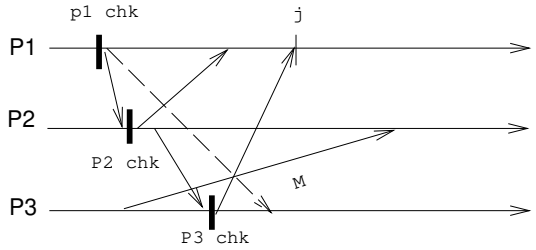

Figure 2: Message losses due to a site failure

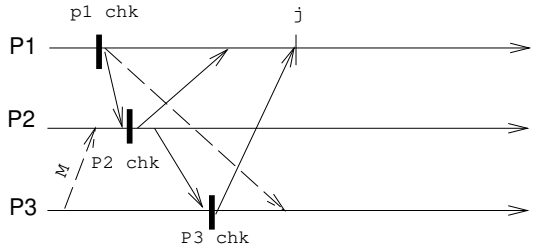

Figure 3: Message losses due to a channel failure

Another error could occur if message M is lost during delivery, see Figure 3: If the channel is temporarily broken, and message M is lost in transit, process P_2 will never obtain that message.

In order to guarantee a correct recovery following a failure, the coordinator must not commit a consistent checkpoint unless it is informed not only that all the tentative checkpoints have been taken, but also that there are no messages in transit[2] or lost. The key issue here is how to implement this efficiently. In the following sections we propose two mechanisms to deal with this issue.

4.2.1 Mechanism 1

This mechanism is derived from the approach by Mattern[18]. It differs from Mattern's system in two respects: 1. Mattern assumes that no messages are lost during delivery, whereas our mechanism tolerates message losses

[2]A message is *in transit* if it is sent before the sender takes the current checkpoint, and is received after the receiver takes the current checkpoint.

during delivery using a timeout technique; 2. while Mattern uses a one-phase commit protocol to take a consistent checkpoint, we employ a variant of two-phase commit without increasing communication overhead while making more efficient use of stable storage (see the previous section).

The mechanism is based on the following three techniques:

1. Like Mattern, we use a *message transit vector* (MTV) to determine the number of messages in transit and detect message losses. Within a distributed shared memory application, processes communicate with each other through the DSM runtime system; instead of each process having its own MTV, the FTSM on each node keeps its MTV. The MTV is a vector of length n, the number of nodes in the distributed system. When node i sends a message to node j, it increments the jth component of the local MTV: $MTV_i[j] = MTV_i[j] + 1$. Node i decrements the ith component of its local MTV, $MTV_i[i] = MTV_i[i] - 1$, whenever a message arrives from another node. In our approach, each FTSM has two MTVs – Pre_MTV and Cur_MTV. If a node sends a message before taking tentative checkpoints, it will modify its Pre_MTV. Similarly, the receiver will modify its Pre_MTV whenever it receives a message. Otherwise, if a message was sent after a tentative checkpoint has been initiated, the sender and receiver will modify their Cur_MTV. After the tentative checkpoints on a node become permanent, the value of the Cur_MTV is copied to the Pre_MTVand the Cur_MTV is cleared. In detail, the message transit vector works as follows:

   - the sender of a message will modify its Pre_MTV if the status bit is set, otherwise the Cur_MTV;
   - if a message is received and its CSN agrees with the local CSN, the Cur_MTV is modified if both, the message and the local status bit are unset, otherwise the Pre_MTV is modified;
   - if a message is received with a CSN less than the local one, the message is logged and the Cur_MTV is updated;
   - if a message is received with a CSN greater than the local one and the local status bit is not set, the current tentative checkpoints are made permanent. Irrespective of the status bits, the local FTSM is then notified to take new tentative checkpoints, which results in the CSN being incremented. The Cur_MTV is then updated;
   - when a node receives a checkpointing request (explicitly or implicitly), the FTSM checkpoints the local processes and replies to the coordinator with a message tagged with its local Pre_MTV;
   - Once the coordinator is informed that all the processes have been checkpointed and there are no messages in transit, it commits the current consistent checkpoint by setting the status bit and making the local tentative checkpoints permanent. There are no messages in transit if

     $$Sum\_V := \sum_{i=1}^{n} Pre\_MTV_i$$

     is a vector of all zeros.

   If all processes have been checkpointed and Sum_V is not zero, there must be outstanding messages. There are two options for the coordinator to proceed:

   - a *pessimistic strategy* is based on the assumption that there are usually some messages in transit. If Sum_V is not zero, the coordinator sends a message to each node i (except itself) where $Sum\_V[i] \neq 0$. This message is tagged with $Sum\_V[i]$, which is the number of outstanding messages to node i. When node i receives this message, it will await arrival of all outstanding messages indicated by $Sum\_V[i]$, and then send the updated local Pre_MTV to the coordinator. The pessimistic strategy is suitable for distributed parallel applications in which processes communicate with each other frequently;
   - an *optimistic strategy* assumes that messages in transit are rare. Therefore, the coordinator does not send extra messages to other nodes. Whenever a node receives a message in transit, it modifies the local Pre_MTV and sends the vector to the coordinator. Assume, for example, that after the FTSM on a node checkpoints the local processes and replies to the coordinator, there are still two messages which were sent before the current checkpoint and have not arrived at this node. Two extra messages containing the updated local Pre_MTV will have to be sent to the coordinator when the messages in transit finally arrive.

     Since this method requires one extra message for each message in transit, it is unsuitable for cases where there are usually a large number of such messages. If, however, there is relatively little communication between the processes of a distributed application, this strategy will result in fewer messages than the pessimistic one.

2. *Minimum message logging*: As described above, a received messages needs to be logged if and only if its

CSN is less than the local one. This logging is done by the FTSM as part of the current checkpoints. With the pessimistic strategy, the FTSM will reply to the coordinator with the updated local Pre_MTV after all the messages in transit as indicated by $Sum\_V[i]$ are received, with the optimistic strategy, the FTSM will reply to the coordinator with the updated local Pre_MTV every time a message in transit arrives.

3. *Timeout mechanism*: If a message is lost during delivery, or if a node fails after it takes the tentative checkpoints and replies to the coordinator while some messages in transit have not arrived at this node, Sum_V will never become zero. In these cases, the checkpointing algorithm cannot terminate (either *commit* or *abort*). We thus use a timeout mechanism to address these problems: The coordinator keeps checking the replies from other nodes and the value of Sum_V. If, within the timeout interval, if all the replies are "yes" and Sum_V becomes zero, the coordinator commits the current consistent checkpoint, otherwise, site failures or message losses are assumed to have occured, and the coordinator aborts the current consistent checkpoint and starts a rollback recovery.

There are several disadvantages to this mechanism:

1. the method used to catch messages in transit causes another round of communication overhead. Moreover, especially when the processes of an application run on many machines (n is large), the MTVs piggybacked on the reply messages can be long;

2. message losses due to channels failures may occur even with a reliable transport protocol. If acknowledgements are not picked up at user-level, message losses cannot be detected until the next consistent checkpoint times out. This may result in a long rollback.

If acknowledgements are picked up at user-level, another method can be used which avoids both of these drawbacks (see next section).

4.2.2 Mechanism 2

In this section, we propose another method to ensure the consistency of the checkpoints in the presence of failures without the problems of mechanism 1. This approach combines the higher level checkpointing algorithms with the underlying transmission protocols. The method consists of two components which work together:

1) A *user-level reliable transmission protocol* (URTP) tailored to our checkpointing algorithms is used. It handles not only data transmission but also message logging (when needed), this allows the higher level algorithms to use a single round of communication for taking a consistent checkpoint (no extra communication overhead is needed for catching messages in transit). Figure 4 shows the configuration of this protocol. Its features are:

- threads are used to provide non-blocking communication;

- if, on receiving a data packet of a message, the CSN of the data packet is less than the local one, the receiver logs the packet and sends an acknowledgement to the sender. If the CSN of the received package is greater than the local one, the receiver informs the local FTSM to take local checkpoints before sending the acknowledgement. As long as the CSNs agree, acknowledgements are sent immediately;

- on sending a data packet of a message, the sender increments the value of a local *acknowledgement counter*. On receiving an acknowledgement, the sender decrements the value of the corresponding acknowledgement counter. If the acknowledgement has not arrived after a certain number of retransmissions, a site failure or a channel failure is assumed to have occurred and the sender sends a *rollback recovery* request to the coordinator.

2) The acknowledgement counter (AC) is used to record the number of message packets not yet acknowledged which were sent from local node to other nodes between two checkpoints. On each node, the FTSM maintains two ACs: previous AC (PAC) and current AC (CAC). If a node sends a packet before taking the current checkpoint, the PAC is modified, otherwise the CAC. After the tentative checkpoints on a node become permanent, the value of PAC is set to that of CAC, and the value of CAC is set to zero. As described above, the value of an AC is incremented when a packet is sent and is decremented when the packet is acknowledged. Notice that ACs just contain local control information and do not need to be transferred among the nodes. With ACs, our checkpointing algorithms work as follows:

- After a FTSM takes its local checkpoints, it will not reply to the coordinator until *its local PAC becomes zero*, at which time it is certain that all the message packets sent *from this node to any other nodes* between the last two checkpoints have arrived at their destinations and have been logged if necessary.

- If the coordinator receives the replies from all other nodes within the timeout interval, it knows not only that all the processes within the application have been checkpointed, but also that there are no messages in transit. It can therefore commit the current consistent

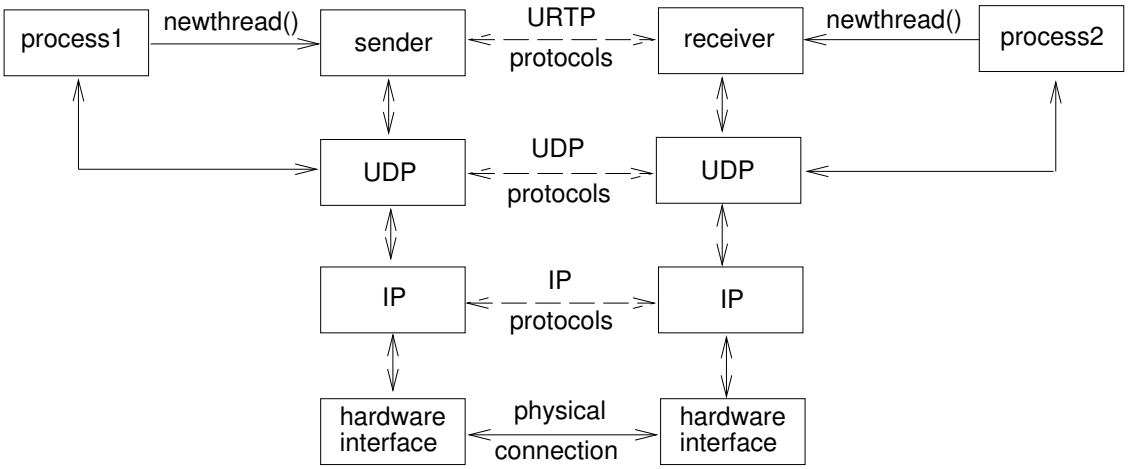

Figure 4: The configuration of the URTP protocols

checkpoint, otherwise, a failure is assumed to have occurred.

Compared to the first method, the detection of a message loss will not be delayed until the next consistent checkpoint while this method only needs one round of communication to take a consistent checkpoint, and does not involve the overhead produced by the message transit vector. Like the first method, this method also needs to use message logging and timeout techniques.

4.2.3 Checkpointing algorithms

Here we present our checkpointing algorithms corresponding to the mechanisms described above.

1) The coordinator:

```
Begin the checkpointing operation;
CSN++;
Checkpoint(local processes of application);
Multicast(chkp_req);
/* waits for the replies from other FTSMs */
if (!timeout &&all FTSMs replied 'y') then
#ifdef MECHANISM 1
   if (Sum_V != 0) then
#ifdef PESSIMISTIC
      for i = 1 to n
         if (Sum_V[i] != 0) then
            send(i,Sum_V[i]);
#endif /* PESSIMISTIC */
      while (!timeout && Sum_V!=0)
         wait;
      if (Sum_V != 0) then abort;
#endif /* MECHANISM 1 */
   commit;
else /* assume failures */
   abort;
```

2) The other working nodes:

```
if (checkpoint request) then
   CSN++;
   modify local control information;
   Checkpoint(local processes of application);
#if MECHANISM 2
   while (PAC!=0)
      wait;
   send(coordinator, 'y');
#endif /* MECHANISM 2 */
#if MECHANISM 1
   send(coordinator,'y',Pre_MTV);
#endif /* MECHANISM 1 */
```

5 Rollback recovery

When failures are detected, failed processes can be relocated to a working node, their states can be recovered from their checkpoints stored on stable storage.

In order to avoid *livelocks* and maximise the parallelism during a rollback recovery, like checkpoints, each rollback recovery is uniquely identified by an increasing *recovery sequence number* (RSN). Each normal message is tagged not only with the CSN, but also the RSN of the sender.

In principle, after determining that the ith checkpoints form the latest consistent state of the application, the coordinator broadcasts marker messages—rollback recovery requests—containing the current RSN to all other nodes, and makes the processes on the local node restart from their ith checkpoints. When a node receives such a marker message, or any message whose RSN is greater than the local one, the FTSM on the node makes the local processes restart from their ith checkpoints, and sends an acknowledgement

to the coordinator. When the RSN of an incoming message is less than the local one, this message must be discarded because it was sent before the current rollback recovery started. With this one-phase commit protocol, our approach implements rollback recovery efficiently.
According to the above description, the **coordinator** works as follows:

```
Begin the rollback recovery operation;
if (local status bit == 1) then
   rollbackto = CSN
else
   rollbackto = CSN −1;
RSN++;
Multicast(RSN, rollbackto);
restart (rollbackto);
if (!timeout && all FTSMs replied 'y') then
   commit
else
   abort; /* assume failure */
```

Other working nodes work as follows when they are informed (explicitly or implicitly) to start a rollback:

```
if (rollback recovery request) then
   RSN = request.RSN;
   CSN = request.rollbackto;
   Rollback(CSN);
   send (coordinator,...);
```

6 Conclusion

This paper proposes an approach for adding fault tolerance, based on consistent checkpointing, to distributed shared memory applications. Compared to other systems, our approach has the following features: 1. two different mechanisms are provided to efficiently address the issue of message losses due to either site failures or unreliable Non-FIFO channels; 2. a variant of two-phase commit protocol is employed keeping the communication overhead the same as that of systems using a one-phase commit protocol while utilising stable storage more efficiently.

Acknowledgements

We would like to thank Anne Ngu, Jayasooriah, Toong Shoon Chan and the anonymous IWOOOS referees for their helpful comments on an earlier version of this paper.

References

[1] S. Adve and M. Hill. Weak ordering – A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture,* pages 2–14, May 1990.

[2] M. Ahuja. Flush primitives for asynchronous distributed systems. *Information Processing Letters* 34, pages 5–12, 1990.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[4] J.B. Carter. Efficient distributed shared memory based on multi-protocol release consistency. *Ph.D thesis,* Rice University. September 1993.

[5] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Systems,* vol. 3, no. 1, pages 63–75, February 1985.

[6] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessor. *IEEE Trans. Software Engineering,* 16(6):660-673, June 1990.

[7] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems,* pages 39–47, October, 1992.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture,* pages 15–26, May 1990.

[9] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message Logging and checkpointing. *Journal of Algorithms,* vol. 11, pages 462–491, 1990.

[10] D.B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proceedings of the 12th Symposium on Reliable Distributed Systems,* pages 86–95, October 1993.

[11] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture,* pages 13–21, May 1992.

[12] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Software Eng,* vol. 13, no. 1, pages 23–31, January 1987.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers,* C-28(9):241-248, September 1979.

[14] T.H. Lai and T.H. Yang. On distributed snapshots. *Information Processing Letters,* 25, pages 153–158, May 1987.

[15] K. Li, J.F. Naughton, and J.S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming,* pages 79–88, March 1990.

[16] K. Li, J.F. Naughton, and J.S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 8, pages 874–879, August 1994.

[17] K. Li, J.F. Naughton, and J.S. Plank. Checkpointing multicomputer applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems,* pages 66–75, September 1991.

[18] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing,* vol. 18, pages 423–434, August 1993.

[19] L.M. Silva and J.G. Silva. Global checkpointing for distributed programs. In *Proceedings of the 11th Symposium on Reliable Distributed Systems,* pages 155–162, October 1992.

[20] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Systems,* vol. 3, no. 3, pages 204–226, 1985.

[21] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing,* August, 1989.

[22] T.J. Wilkinson. Implementing fault tolerance in a 64-bit distributed operating system. *Ph.D thesis,* City Univ., London, July 1993.