Rollback and Recovery Mechanisms

In Distributed Systems

By

Priya Venkitakrishnan

Department of Computer Science, University of Texas at Arlington

Abstract

This paper explains rollback and recovery mechanisms in distributed systems. Two different approaches to checkpoint creation and roll back recovery are the prime focus of this paper. The different algorithms used by each approach are discussed. A comparison of the two approaches is made at the end. The paper also brings to light some of the other techniques for roll back and recovery that have been proposed.

1 Introduction

In a distributed system, hundreds of users concurrently perform various database transactions. Different servers carry out these transactions in the distributed system and the results are returned to the user. A flight reservation system, banking or stock markets all are typical examples of distributed systems. Two main properties that distributed transactions must have are atomicity in the presence of failures and durability [5]. Atomicity of a distributed transaction ensures that the transaction is either carried out completely without any failure, or if a failure occurs, the transaction is fully discarded. Durability ensures that the results of a transaction are recorded in permanent storage and may be made available at any future time. In order to achieve the above-mentioned properties, each system is provided with a recovery manager [5], whose main goal is to save the effects of transactions in permanent storage or a recovery file and restore the server to a consistent state when a failure occurs. A transaction at a lower level may be termed a process that is carried out by a server. In this paper when we refer to a 'process' it necessarily means a transaction from a user's point of view.

Checkpoint and Roll back recovery are the two important techniques used by a recovery manager to recover the state of a process in the event of failure and let the process proceed normally, in spite of failure. A checkpoint is an entry made in the recovery file at specific intervals of time that will force all the currently committed values to the stable storage. Whenever a process fails it will roll back to the most recent checkpoint made and restart the system from a previously consistent state. In a distributed system, the recovery managers need to make sure that these checkpoints lead the system to a globally consistent state when a server recovers from a failure and restarts.

This paper will address two different approaches for checkpointing and roll back recovery in distributed systems. The paper is organized as follows: in section 2, the notion of global consistency in distributed systems and the two main problems associated with checkpointing and roll back recovery are explained. In sections 3 and 4, two different approaches to checkpointing and roll back recovery in distributed systems are discussed. The two approaches are compared and contrasted in section 5. Section

6 gives a brief overview of various other mechanisms used for rollback and recovery in distributed systems. Section 6 also presents inferences and conclusions.

2 Consistency in a Distributed System – The Domino Effect and Livelock Problem

There are two main approaches for creating checkpoints in a distributed system. In the first approach, every process takes checkpoints independently and the currently committed results are stored in permanent storage. When one or more of the processes fail, they need to communicate with other processes in the system to find a consistent set of checkpoints among the saved ones. All the affected processes are rolled back to this set of checkpoints and then restarted.

In the second approach, every process coordinates with all other processes in the system before it takes a checkpoint and hence at any instant the set of checkpoints are guaranteed to be consistent.

Domino effect may occur in the first approach as explained below. Assume two processes in the system, X and Y that have independently taken checkpoints as shown in Figure 1[1]

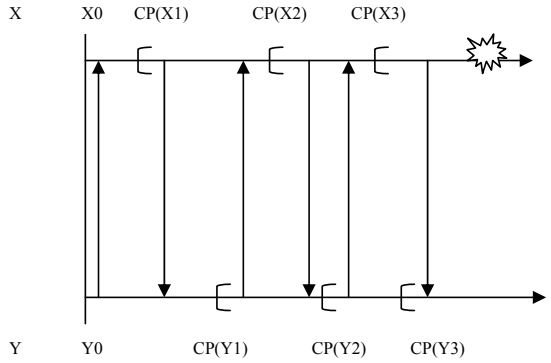

Figure. 1: Interleaved Checkpoints taken by two processes [1]

When process X fails at the point shown, it will roll back to the checkpoint CP(X3) shown in Figure 1. Now process Y has the record of a receipt of a message, which is undone or rolled back in X. So Y will also roll back to discard the effect of the undone process, to CP(Y2). As a result of this, the message sent by Y is rolled back. This leaves X with an undone process and X will also roll back. This will continue until X and Y are rolled back to their respective beginning checkpoints, which results in a globally consistent state. This effect called the "**Domino effect**" is very undesirable, as it will lead to unnecessary delays in the overall completion of the entire transaction.

The Livelock problem may arise when a process rolls back to its checkpoint after a failure and requests all the other affected processes also to roll back. In such a situation if the roll back mechanism has no synchronization, it may lead to the **livelock** problem as described below [1]. Assume two processes X and Y. Consider the following sequence of operations:

1. X sends a message M1 and fails before it receives the message M2 from Y.
2. X rolls back to its recent checkpoint and recovers; it receives M2 from Y and sends M3 to Y.
3. X has no record of sending M1 while Y has a record of its receipt. Hence Y also rolls back and notifies X.

4. The global state is inconsistent, as Y has no record of sending M1 while X has a record of its receipt.

In this manner, both X and Y may be forced to roll back forever even though no failures occur [1]. These indefinite rollbacks are caused due to the lack of synchronization in the checkpoint creation among various processes involved in a distributed transaction.

It is very important that the notion of consistency be maintained in distributed systems, which involves multiple transactions taking place at different servers. A global state consists of a set of local states from each process in the system and may be represented as: [6]

GS = {LS1, LS2, LS3, … LSn}

GS is the global state
LS1, LS2.. LSn define the local states.

The global state is said to be consistent if for every message M1, sent from one state to another, there is a corresponding receive message on the receiver. If all the checkpoints in the local state have recorded these events, then the system is in a globally consistent state. To achieve this goal, the two main issues are checkpoint creation and roll back recovery. In other words, it is very important to know when the systems take checkpoints, the state of results stored in stable storage and how a process recovers in the event of failure.

The two approaches discussed below, address these issues and make an attempt to avoid the Domino effect and the Livelock problem.

3 Checkpointing and Rollback-Recovery – A Synchronized Approach

In this algorithm proposed by Richard Koo and Sam Toueg [1], processes take checkpoints in a coordinated manner. Before we describe the algorithm let us define briefly a few assumptions relevant to this algorithm.

In this approach, it is assumed that processes communicate by exchanging messages through communication channels and these channels are designed in a first in first out manner (FIFO). Also, processes do not share a common memory or clock. When a process fails all other processes are made aware of this failure in a fixed time interval. Problems such as partition of network due to failures are not considered.

As mentioned above, the main aim of a checkpointing and roll back approach is to maintain a globally consistent state among various processes involved in the system. “A set of checkpoints, one per process in the system is said to be consistent if the saved states form a consistent global state” [1].

3.1 Checkpoint Algorithm

In Koo’s and Toueg’s approach [1], checkpointing is done in a pattern similar to a two phase commit protocol.

Phase1: In the first phase, the initiator takes a tentative checkpoint and sends a request to all processes involved in the transaction to take a tentative checkpoint. Each of the participating processes sends a decision to the initiator. The initiator decides either all tentative checkpoints including its own

tentative checkpoint to be made permanent or discarded, based on the response from each participating process.

Phase 2: In the second phase, the initiator sends its decision to all related processes. When a process receives this request, it either makes its tentative checkpoint permanent or discards the checkpoint.

In addition to following a two phase commit pattern, the algorithm also requires only a minimal number of processes to take checkpoints. For this, one more condition is enforced on each participating process as mentioned below.

a) Every process receives a checkpoint request from an initiator.
b) Process X will only take a checkpoint if the tentative checkpoint taken by the initiator has recorded the receipt of a message M1 from that process and the last permanent checkpoint made by X has not recorded the sending of this message M1 to the initiator.

Let us take an example and explain this checkpointing algorithm. Consider four processes A, B, C, and D involved in a distributed transaction. The points where each of them takes a checkpoint during the course of the transaction is shown in Figure 2[6]. Also, assume at time T1 process A wants to take a checkpoint.

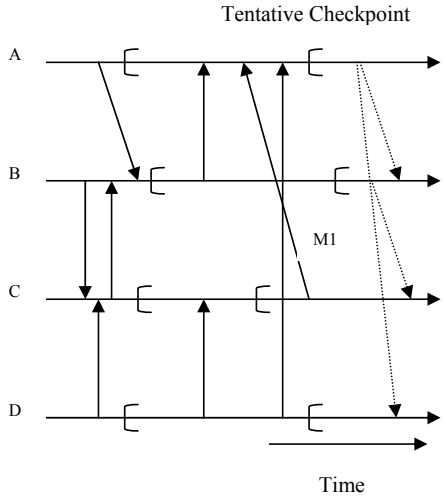

Figure 2: Checkpoint Creation using algorithm in [1]

Process A will take a tentative checkpoint and send requests to processes B, C and D. The requests are indicated as dotted lines. From the figure it is clear that processes B and D will not take a tentative checkpoint but process C will agree to take a tentative checkpoint since the sending of message M1 is recorded in the tentative checkpoint of A but is not yet recorded in the last permanent checkpoint C. Hence in this example the initiator, process A will decide not to checkpoint at that instant of time and will send its decision to processes B, C and D.

Koo and Toueg [1] have suggested a labeling scheme to implement this algorithm, which can be briefly described as follows:

Each process attaches a monotonically increasing label to each outgoing message. $\top$ is defined as the largest label. $\perp$ as the smallest label. The last message that a process X received from process Y, after taking a permanent or tentative checkpoint is denoted by **last_recmsg$_x$(Y)** and the first message that a process X sent to process Y is denoted by **first_sndmsg$_x$(Y)** . Whenever a process X wants to take a checkpoint it sends its request to all processes in the system along with the label of **last_recmsg$_x$(Y)** . Process Y takes a checkpoint only if:

last_recmsg$_x$(Y) >= first_sndmsg$_y$(X) > $\perp$. [1]

A variable willing_to_ckpt is associated with every process [1]. When a process is willing to take a checkpoint, it sets willing_to_ckpt to "yes" and sends its reply to the initiator. If a process is unwilling to take a checkpoint it sets willing_to_ckpt to "no". The algorithm also restricts the participating processes from sending any normal messages from the time it takes a checkpoint to the time it receives the decision from the initiator.

3.2 Rollback Recovery Algorithm

The roll back recovery algorithm is based on a pattern similar to the two-phase commit protocol. When a failure occurs in a process, the process recovers or rolls back to a previously consistent state and sends a request to all other processes to restart. The initiator makes a final decision as to which processes need to restart based on the response from all processes in the system. During the second phase of the roll back recovery algorithm, the initiator sends its final decision to all processes and all processes carry out the corresponding decision.

The algorithm causes only a minimal number of processes to roll back, by imposing an additional constraint. A process Y will roll back based on a request from the initiator X, only if X's rollback undoes the sending of a message to Y [1]. Like the Checkpoint algorithm, the roll back recovery algorithm is also implemented using a labeling scheme.

Let **last_sndmsg$_y$(X)** be the label of the last message process Y sent to X and **last_recmsg$_x$(Y)** be the label of the last message process X received from Y. When process X receives a rollback request from Y, X will rollback only if:

last_recmsg$_x$(Y) > **last_sndmsg$_y$(X)**. [1]

4 Optimistic Algorithm for Independent Checkpointing and Concurrent Rollback

In Independent Checkpointing, processes take checkpoints independently and the already committed results are stored in permanent storage. Here there is no coordination of processes involved before they take the checkpoints. When some failure occurs, processes need to communicate with each other to determine a consistent set of checkpoints [2]. As a result of this, in Independent checkpointing there are greater chances for the Domino effect to occur.

In the algorithm proposed by Bharat Bhargava and Shy-Renn Lian [2], a recovering process computes a globally consistent set of checkpoints and this reduces the overhead involved in maintaining a globally consistent state even when there are no failures. Before we discuss the algorithms for independent checkpointing and rollback recovery, we will define the system model, the terminology used to describe the algorithm and data structures involved.

4.1 Terminology and System Model [2]

- **System Model** [2]: The processes do not share memory and communicate using dedicated first in first out channels. It is also assumed that when a process fails, other processes are informed about the failure within a fixed interval of time.

- **Checkpoint Number** [2]**:** All checkpoints taken by processes are given sequence numbers and are incremented every time a process takes a checkpoint. cn_p [2] denotes the sequence number of a checkpoint taken by process P. p^{cnp} denotes the checkpoint whose checkpoint number is cn_p.

- **Checkpoint Interval** [2]: The interval between two consecutive checkpoints. For the above notation the interval between p^{cnp-1} and p^{cnp} is called the checkpoint interval of p^{cnp}. A virtual checkpoint p^f is defined for the checkpoint interval between the current instant and the last checkpoint. This virtual checkpoint will help processes to record all the messages received during the current instant and the time when last checkpoint was taken.

- **Operational Session** [2]: The interval of time between the start of a process and the time it fails. For a process p the operational session is indicated by sn_p and is incremented each time a process rolls back after a failure and resumes execution.

- **Input Information Table** [2]: Every process maintains an Input Information table denoted by IIP $_{(process\ name)}$. The IIP of a process P may be represented as a two-dimensional array as shown in Figure 3. The columns represent the processes in the system and the rows represent the various checkpoint intervals of the process P. Each entry in the table denotes for a given checkpoint interval of P, the messages received by P from other processes in the system. The checkpoint intervals at which these messages are sent from other processes is also recorded in the Input Information Table.

Checkpoint Interval	Process Q	Process R	Process S
p^{cnp}	$m_{qp}(x)$-q^{cnq}, $m_{qp}(y)$ - q^{cnq}		$m_{sp}(z)$ - s^{cns+2}
p^{cnp+1}		$m_{rp}(x)$ - r^{cnr}	
p^{cnp+2}		$m_{rp}(y)$ - r^{cnq-1}	

Figure 3: Input Information Table for Process P

From the figure it can be seen that during checkpoint interval p^{cnp+1}, process R has sent a message $m_{rp}(x)$, during checkpoint interval p^{cnp} , process Q has sent messages $m_{qp}(x)$ and $m_{qp}(y)$ and process S has sent $m_{sp}(z)$. It can also be noted that $m_{qp}(X)$ was sent during checkpoint interval q^{cnq} of Q, $m_{rp}(X)$ during checkpoint interval r^{cnr} of r and so on.

- **System Messages** [2]
  Any system message consists of:
  1. Name of the sending process
  2. Name of receiving process
  3. Current session number
  4. Current checkpoint number

5. Message content

System messages may be normal system messages or control messages. There are three types of control messages defined:

- **Rollback_initiating request** [2]: This message is sent by a recovering process to all other processes.
- **Rollback_request message** [2]: The rollback request message contains the final decision made by the rollback initiator. Rollback initiator is a process that has encountered a failure and is trying to recover.
- **Input information message** [2]: Every process sends an input information message in response to a rollback_initiating request from the initiator. The input information message is constructed from the Input information table described above. The input information message a process q sends to a rollback initiator p consists of all the checkpoint intervals of q and all the messages that q received from all processes in the system during those checkpoint intervals.

  Every process also makes a mark in the input information table, indicating the point up to which it has previously sent the input information message. For any recurring roll rollback_initiating request, the process will only send checkpoint information made after the mark. This will prevent the processes from sending redundant information to the initiator. [2]
- **Local graph** [2]: A rollback initiator constructs the local graph using the input information messages. The nodes of the graph represent the checkpoint numbers of other processes known by the local process. The graph has two different types of edges. Edges, which connect checkpoints taken by the same process, and those, which represent the message flow between processes during that checkpoint interval. Every process first constructs the local graph when it fails for the first time and from then on, the initiator updates the local graph based on the current input information message.

4.2 Independent Checkpoint Algorithm

Every process takes checkpoints at fixed intervals of time. For a process P the Independent Checkpoint algorithm is executed as follows [2]:

- Initialize Checkpoint number cn_p to 0
- Initialize fixed time interval to cp_time
- Initialize count of local clock tik to local_clock_tik
- While local_clock_tik < cp_time
  Execute normal operations
  Increment clock_tik
- When local_clock_tik > cp_time
  Increment cn_p

4.3 Roll Back Algorithm

The roll back algorithm for a process that fails is executed in a manner similar to the two-phase commit protocol. The steps involved in the rollback recovery algorithm are given below [2].

Phase 1:

1. Process p rolls back to its most recent checkpoint p^{cnp}
2. p increments its operational session number, sn_p.
3. p sends rollback_initiating request to all other processes in the system .
4. p collects the input information messages from all other processes

Phase 2:

1. Process p constructs a local system graph
2. The system graph is traversed by p to find out the processes that need to rollback and the checkpoints they need to rollback to if necessary.
3. Process p sends its final decision to all processes in the rollback request message.

The rollback algorithm can be explained using an example. Consider a system consisting of three processes, P, Q, R and S. The snapshot of the system was taken at a given instance of time, when process P has encountered a failure. Process P rolls back to its most recent checkpoint and sends rollback_initiating request to processes Q, R and S in the system.

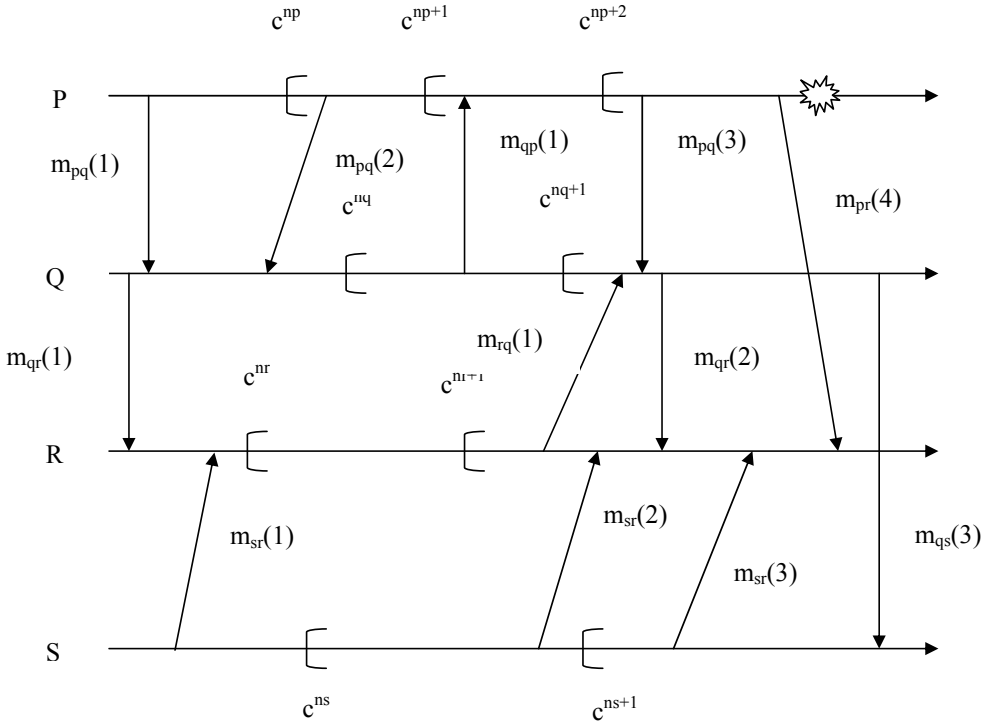

Figure 4: Snapshot of a distributed system when process P has failed

The following figures show the Input Information table for each process P, Q, R and S.

Process P

Checkpoint Interval	Process Q	Process R	Process S
p^{cnp}			
p^{cnp+1}			
p^{cnp+2}	$m_{qp}(1)$ - q^{cnq+1}		
p^{f}			

Figure 5: Input Information Table of Process P

Process Q

Checkpoint Interval	Process P	Process R	Process S
q^{cnq}	$m_{pq}(1)$ - p^{cnp}, $m_{pq}(2)$ - p^{cnp+1}		
q^{cnq+1}			
q^{f}	$m_{pq}(3) – p^{f}$, $m_{rq}(1) – r^{f}$		

Figure 6: Input Information Table of Process Q

Process R

Checkpoint Interval	Process P	Process Q	Process S
r^{cnr}		$m_{qr}(1)$ - q^{cnq}	$m_{sr}(1)$ - s^{cns}
r^{cnr+1}			
r^{f}	$m_{pr}(4) – p^{f}$	$m_{qr}(2) – q^{f}$	$m_{sr}(2) – s^{cns+1}$, $m_{sr}(3) – s^{f}$

Figure 7: Input Information Table of Process R

Process S

Checkpoint Interval	Process P	Process Q	Process R
s^{cns}			
s^{cns+1}			
s^{f}		$m_{qs}(3) – q^{f}$	

Figure 8: Input Information Table of Process S

Formatted

The rollback initiator process P gets the input information messages from all other processes, Q, R and S. Process P will then construct a local system graph with the input information messages from all processes. If the graph had been constructed previously, the initiator will only update the graph with the currently received message flow information. To determine the processes that need to roll back and the checkpoints they need to rollback to, the initiator runs a depth first search on the local graph.

For the above example, mentioned, it can be seen that when process P recovers it will roll back to its most recent checkpoint c^{np+2}. It will then send a rollback_initiating request to processes Q, R and S. P receives input information messages from all processes. Process P will then construct the local graph and determine the roll back points for processes Q, R and S. Process P will then send the following rollback request messages to processes Q, R and S

Process Q: Roll back till cn^{q+1} since during the checkpoint interval pf, process Q has received $m_{pq}(3)$ from Process P and during checkpoint interval r^f, process Q has received $m_{rq}(1)$.

Process R: Roll back to cn^{r+1} since during checkpoint interval p^f, process R has received $m_{pr}(4)$ and during checkpoint interval qf, process R has received $m_{qr}(2)$.

Process S: Roll back to cn^s. The reason is since process S has receives $m_{qs}(3)$ during checkpoint interval s^f and will first roll back to cn^{s+1}. Again since process R has rolled back to cn^{r+1} and process R has received $m_{sr}(3)$ and $m_{sr}(2)$ during checkpoint interval r^f, process S has to roll back to cn^s.

From the point processes Q, R and S sends the input information message to the point it receives the final decision from the initiator, Q, R and S will not send any normal messages but will receive normal system messages from other processes. When they receive the roll back request message from P they will check if the rollback set contains any of the processes from which it received the message. In such a situation, the processes Q, R and S will restart from the point they sent the input information message and continue their normal operations. [2]

5 Comparison of the two Approaches

The main goal of the checkpointing and roll back algorithms discussed above is to restore a distributed system to a globally consistent state after transient failures [1]. In the first approach [1], the creations of checkpoints are done in a coordinated manner. One of the main advantages of this approach is that it avoids the Domino effect. Since the checkpoints are taken by effectively communicating with all processes, the algorithm also reduces the livelock problem, related with rollback recovery [1]. In the two-phase approach followed by this algorithm, it requires only the related processes be part of the checkpointing and roll back recovery algorithm.

In the coordinated approach, a globally consistent state is always maintained in the distributed system. When a process fails, the coordinator communicates with other processes and restores the system back to a globally consistent state. In order to maintain this globally consistent state, the processes require a number of messages to communicate between each other to create checkpoints. As a result of this, some synchronization delays may be introduced in the system [2]. Such delays are very critical to real time applications [3]. The worst-case time for the completion of this algorithm is dependent on the number of processes in the system and to the upper bond on the communication delay between any two processes [9]. A variation of the approach is suggested in [3] to avoid delay due to coordination. In [3], each participant can make their own decision by communicating with other dependent processes. There is no Phase 2 involved in the algorithm suggested in [3].

In the Independent checkpointing approach, the cost of creating checkpoints is low [3]. Each process has maximum autonomy in deciding when to take checkpoints [7]. But this may lead to processes taking useless checkpoints that will never be part of a global checkpoint. Such checkpoints will also produce additional storage overhead [7]. These useless checkpoints need to be removed from the stable storage using some garbage collection mechanism [7]. When a process fails the processes communicate with each other to produce a consistent set of checkpoints. Problems such as the Domino effect or Livelock problem may occur. In this method, a worst-case scenario would be for processes to roll back to their initial states and restart their computation from the beginning [4]. Also, in this approach suggested in [2], a process, after it rolls back may receive messages that are undone by the sender. This will cause the process to roll back again and again and might result in a cyclic restoration effect [4]. This can be avoided by not permitting a rolled back process to receive or send messages to other processes until the rollback recovery algorithm is terminated [4].

6 Conclusion

In addition to some of the approaches mentioned above, there are several other mechanisms suggested for roll back and recovery in distributed systems. In [8] different protocols are mentioned for independent recovery in a large-scale distributed system which has replicated data. These protocols ensure consistency among the various replica by using a log based recovery approach and also storing the before image and the after image of the transactions in their log. In the method suggested by [9], individual processes coordinate using a common time base before saving their state to stable storage. This is achieved by synchronizing the clocks of all processor's in the system.

Today, distributed systems are becoming ubiquitous and enable many applications to communicate over an interconnected network. Distributed systems are employed in almost all sectors such as banking, reservation systems and in many other business environments. One of the main problems faced by these large systems is their susceptibility to failure. The various checkpointing and rollback recovery mechanisms are quite effective and will definitely make distributed systems a reliable and efficient environment for different applications.

7 References

[1] Richard Koo and Sam Toueg , "Checkpointing and Rollback Recovery for Distributed Systems" , Proceeding of 1986 Fall Joint Computer Conference, 1986, Dallas, Texas

[2] Bharat Bhargava and Shy-Renn Lian, "Independent Checkpointing and Concurrent Rollback for recovery in distributed systems – an Optimistic approach", Proceedings on Seventh Symposium on Reliable Distributed Systems, 1988

[3] Junguk L.Kim and Taesoon Park, "An Efficient Protocol for Checkpointing Recovery in Distributed Systems", IEEE Transactions on Parallel and Distributed Systems, Vol.4, No.8, August 1993

[4] Pei-Jyun Len and Bharat Bhargava, "Concurrent Robust Checkpointing and Recovery in Distributed Systems", Fourth International Conference on Data Engineering, 1988. Proceedings.

[5] George Coulouris, Jean Dollimore, Tim Kindberg - Distributed Systems Concepts and Design Addison Wesley Third Edition

[6] Lecture slides on Checkpointing and Recovery (II) http://courses.ece.uiuc.edu/ece442/lecture_22.pdf

[7] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang and David B.Johnson , “A Survey of Rollback-Recovery Protocols in Message-Passing Systems” Draft.

[8] Peter Triantafillou, “Independent Recovery in Large Scale Distributed Systems”, IEEE Transactions on Software Engineering, Vol.22, No.11, November 1996

[9] Parameswaran Ramanathan and Kang G. Shin, “Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System”, IEEE Transactions on Software Engineering, Vol.19, No.6, June 1993

[10] Robert E. Strom and Shaula Yemini, “Optimistic Recovery in Distributed Systems”, ACM Transactions on Computer Systems, Vol.3, August 1985

[11] A. Prasad Sistla and Jennifer L. Welch, “Efficient Distributed Recovery Using Message Logging”, ACM Transactions. Computer Systems, Vol.3, no.3, August 1985

[12] Tony T-Y. Juang and S.Venkatesan, “Crash Recovery With Little Overhead”, Seventh Symposium on Distributed Computing Systems, 1991

[13] Tong, Z and Kain, R.Y and Tsai, W.T., “Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks”, IEEE Transactions on Parallel and Distributed Systems, Volume 3 Issue:2, 1992

[14] Sang Hyuk Son and Ashok K. Agarwala, “Distributed Checkpointing for Globally Consistent States of Databases”, IEEE Transactions on Software Engineering, Vol.15, No.10, October 1989

[15] Avraham Leff and Calton Pu, “A Classification of Transaction Processing Systems”, IEEE Computer

[16] M.Tamer Ozsu and Patrick Valduriez, “Distributed Database Systems: Where Are We now?” IEEE Computer