CRASH RECOVERY WITH LITTLE OVERHEAD
(Preliminary Version)

Tony T-Y. Juang and S. Venkatesan

Computer Science Program, MP 31
University of Texas at Dallas
Richardson, TX 75083-0688
{juang,venky}@utdallas.edu

ABSTRACT

Recovering from processor failures in distributed systems is an important problem in the design and development of reliable systems. Several solutions to this problem have been presented in the literature. Most of them recover from failures by storing sufficient extra information in stable storage and using this information when there are failures. In this paper, we present two solutions to this problem which involve very little overhead. Without appending any information to the messages of the application program, we show that it is possible to recover from failures using O(|V||E|) messages where |V| is the number of processors and |E| is the number of communication links in the system. The second algorithm can be used to recover from processor failures without forcing non-faulty processors to roll back under certain conditions. With a small modification, the second algorithm can also be used to recover from processor failures even if no stable storage is available.

1. INTRODUCTION

Distributed systems are becoming popular because of several advantages they have over centralized ones. The advantages include efficient utilization of resources, ability to enhance the system gradually, greater degree of fault-tolerance, etc. An important and desirable property of a distributed system is its ability to tolerate failures. As the size of distributed systems grows, so does the probability that some component may fail. Thus, it is important to deal with failures of the components of the system. Fault tolerance is provided at two levels of the system -- at the hardware level and at the protocol level. At the hardware level, components are designed and built with high reliability. Faults that occur in spite of the high reliability of the components are dealt with at the protocol level. Thus, specific steps must be taken at the protocol level to increase the reliability of distributed systems.

Coping with processor failures is hard in solving simple problems (and is impossible in instances such as distributed consensus [3]) even if the processor failure mode is restricted to fail-stop failures, while communication failures are comparatively easier to deal with.

In distributed transaction processing systems, there is a need to recover from processor failures quickly to increase the availability of the system. Checkpointing and rollback recovery is a scheme that is widely used. Each processor locally saves its current state and its history in a stable storage from time to time so that if the processor fails, it can restart from the most recently saved state. This process of saving processor states is called *checkpointing*. For the underlying computation to restart from a *consistent global state*, it may be necessary for some or all of the processors in the system to restart from a processor state that occurred before the latest saved state. This is called *rolling back*. To prevent the *domino effect* and to rollback the processor states to the maximum consistent state, certain additional information is appended to each message of the application program. The reader is referred to [2] for a discussion on consistent states of a distributed computation, [16] for a discussion on repeated global state determination, [17] for domino effect, [5] for maximum consistent states in crash recovery, and [6, 12] for a discussion on appending additional information to application messages to aid in rolling back.

Checkpointing has been widely used and studied by many researchers [2, 5-9, 12-14, 17]. There are two approaches towards checkpointing and crash recovery: the synchronous approach and asynchronous approach. The synchronous approach is to ensure that all processors keep local checkpoints in stable storage and coordinate their local checkpointing actions such that the global checkpoints (the set of local checkpoints) in the system is guaranteed to be consistent [2, 7, 9, 15, 17]. When a failure occurs, processors roll back and restart from their most recent checkpoints. That is part of the recent global checkpoints. While crash recovery is easy and simple in this case, additional messages are generated for each checkpoint, and synchronization delays are introduced

CH2996-7/91/0000/0454$01.00 © 1991 IEEE

during normal operations. If there are no failures, then the above approach places an unnecessary burden on the system in the form of additional messages and delays. Similarly, when a processor rolls back and restarts after a failure, a number of additional processors are forced to roll back with it. The processors indeed roll back to a consistent state, but not necessarily to the maximum consistent state.

In the asynchronous approach, each processor takes local checkpoints independently and a consistent global state is constructed using these local checkpoints during recovery. To aid in crash recovery and minimize the amount of work undone in each processor, all of the incoming application messages are logged by the recipient. Message logging can be performed in two ways and the two schemes are pessimistic and optimistic message logging.

In pessimistic message logging, each application message is synchronously logged to stable storage before it is processed [1, 10]. Thus the stable logged information across processors is always consistent and crash recovery is easy. However, since synchronization is needed between logging and processing each of the incoming messages, this protocol slows down the application computation of each processor. It is easy to see that considerably severe overhead is placed on the system even if there are no processor failures.

On the other hand, optimistic protocols perform message logging asynchronously [5, 6, 12, 14]. In this case, each processor continues to execute normally, and the received messages are logged periodically. In case of failure, any message m sent by a failed processor after its checkpoint will create an inconsistent system state if the recipient of message m does not roll back to a point before message m is received. Intuitively, a *maximum consistent state* which undoes the minimal number of application computation in each processor is needed for recovering from failures and restarting the computation.

The algorithm of Strom and Yemini [14] using asynchronous checkpointing approach and optimistic logging protocol causes a processor to roll back $O(2^{|V|})$ times in the worst case where |V| is the total number of processors in the system. It also needs an exponential number of message exchanges to recover from the failure of one processor. Johnson and Zwaenepoel [5] consider several issues relating to optimistic crash recovery and present algorithms for crash recovery using optimistic message logging protocol. Their algorithms use matrices and hence they cannot be directly implemented in distributed systems. Sistla and Welch [12] present two algorithms using the asynchronous approach to recover from processor failures and restart the computation from a maximum consistent global state. One algorithm requires $O(|V|^2)$ message exchanges when O(|V|) additional information is appended to each application message and the other algorithm uses additional $O(|V|^3)$ messages when $O(1)$ extra information is appended to each message. Juang and Venkatesan [6], present two algorithms using the optimistic approach -- one algorithm that uses $O(|V|^2)$ messages to recover from the failures of any number of processors by adding O(1) additional information to each application message, and another algorithm that uses only O(|V|) messages for ring networks (again, by adding O(1) additional information), and can handle multiple processor failures.

Adding additional information to each application message increases the load on the communication system which degrades the system performance. Also, saving any information in stable storage places a load on the system. In this paper, we present two schemes -- one in which no additional information is appended to the messages of the underlying computation. If there are no processor failures or if processor failures are very rare, then this method is very desirable, since that algorithm uses O(|V||E|) messages in the absence of any additional information. This compares well with the best known scheme of [6] which requires $O(|V|^2)$ messages, but that scheme appends one number to each application message. In the second scheme, no roll back is necessary when O(1) additional information is appended to each application message and no more than two adjacent processors fail. Thus, the second algorithm can be used even if no stable storage is available. In both cases, we present recovery algorithms and formally prove that our algorithms are correct as long as no further failures occur during the recovery algorithm.

The paper is organized as follows: In section 2, the computational model and some definitions are presented; section 3 contains a recovery algorithm when no additional information is appended to the application messages; section 4 presents a recovery algorithm where no processor needs to roll back as long as the above-mentioned two conditions hold good and finally section 5 concludes the paper.

2. SYSTEM MODEL

A distributed system can be viewed as a finite collection of processors which are spatially separated, without shared memory or clock, and which communicate with each other by exchanging messages through communication channels. Channels are assumed to have infinite buffers, to be error-free, and to deliver messages in the order sent. The delay experienced by a message in a channel is arbitrary but finite. Processors directly connected by a communication link are called neighbors.

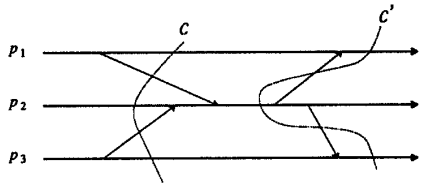

Figure 2.1

As defined in [14], a set of processor states in which each pair of processors agrees on the communication that has taken place between them is called a set of *consistent states*. For example, the time cuts c and c' in Figure 2.1 are consistent and inconsistent cuts respectively. A state of a processor can be lost if the processor fails before that state is saved. If the state of a processor that has sent a message is ever lost, then in order for the system state to be consistent, the state change resulting from the receipt of that message in the receiving processor must be undone; that is, the processor must be *rolled back*. We say that the system is in an *optimum consistent state* if the processor states are consistent and the amount of rollback in each processor is minimum.

To recover from processor crashes and restore the system to a consistent state, we use two types of logs - *volatile log* and *stable log* [4, 10]. Accessing volatile logs requires less time, but the contents of a volatile log are lost if the corresponding processor fails. At irregular intervals, each processor (independently) saves the contents of the volatile log in a stable storage and clears the volatile log, and this is called *checkpointing*. The goal of checkpointing is to eventually log a snapshot of a previous state of the processor in stable storage. We assume that the underlying computation or the application program is *event-driven* where a processor p waits until a message m is received, processes the message m, changes its state from s to s', and sends a (possibly empty) set of messages to some of its neighbors. The new state s' and the contents of the messages transmitted depend on state s (the state of the sender) when m was received and the contents of the message m. For example, in Figure 2.2, processor p_2 changes its state from s_{22} to s_{23} when it processes message m sent by p_3.

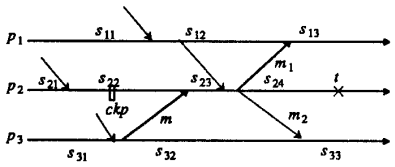

Figure 2.2

Each time a processor receives a message, it begins a new *State Transition Interval* (or STI) which is the interval of time between a processor receiving a message and the time it completes all of the actions associated with processing the message received (including sending messages to its neighbors). Each STI is identified by a unique sequential number called *interval index*, which is simply a count of the number of messages that the processor has received and processed. Since the resulting state of the receiver of a message depends on the state of the sender and the contents of the message, a *dependency* is created by each message. For example, in Figure 2.2, state s_{23} depends on state s_{22} and message m. The contents of m, in turn, depends on state s_{31}. Thus, processor state s_{23} depends on processor state s_{31}. This is an example of a direct dependency. Note that this dependency relation is transitive. In the same example, state s_{13} depends on state s_{23}, and since state s_{23} depends on state s_{31}, processor state s_{13} *transitively depends* on state s_{31}. Several more transitive dependencies can be inferred from Figure 2.2.

Consider the case when p_2 fails at time t as marked in Figure 2.2. Processor p_2 restarts from state s_{22} since that is the latest processor state available in the latest local checkpoint of p_2 (taken at *ckp*). Since the state s_{24} was lost, messages m_1 and m_2 become *orphan* messages. So, p_1 and p_3 both need to roll back to states s_{12} and s_{32}, respectively. In the next two sections, we present recovery techniques that construct consistent global states from which the application program can resume execution, after failures.

3. RECOVERY WITH NO ADDITIONAL INFORMATION

We now present a recovery scheme that works correctly even if no information is added to each application message. This algorithm uses O(|V||E|) messages when an arbitrary number of processors fail where |V| is the total number of processors and |E| is the total number of communication links. If the failures are few and the number of application messages sent is large, this method is preferable. This is because the recovery procedure is run rarely (as processor failures are rare) and no additional load is placed on the communication system as nothing is added to the application messages when the distributed system operates without failures.

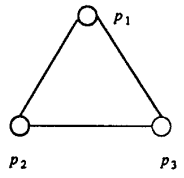

Figure 3.1(a)

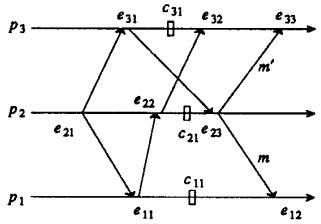

Figure 3.1(b)

Consider a sample network consisting of three processors as shown in Figure 3.1(a). Assume that a

distributed application program is run on this system and let Figure 3.1(b) represent the run of the program. In that figure, e_{ij} represents the j^{th} local event (of the application program) of processor p_i. Similarly, c_{ij} represents the j^{th} checkpoint made by p_i. Assume that p_2 fails. All of the messages that became orphan messages due to direct and transitive dependencies becasue of the failure of p_2 must be identified. Processor p_2 must roll back to the processor state immediately after event e_{22}. This implies that for consistency of the application program, p_1 and p_3 must roll back to processor states after events e_{11} and e_{32} respectively. For p_1 and p_3 to roll back correctly, p_2 must inform p_1 and p_3 that messages m and m' are orphan messages. To correctly identify the orphan messages in the absence of additional information being appended to the application messages, we use the following strategy: Note that every processor has a record of its complete behavior from the beginning to its latest checkpoint. Thus, processor p_2 can determine when it failed with respect to the number of messages it sent to p_1. In Figure 3.1(b), it is clear that if only p_1 and p_2 are under consideration, then as far as rolling back processor p_1 is concerned, p_2 failed after sending the first message to p_1. Thus, p_2 can inform p_1 that the second message from p_2 to p_1 is an orphan message. This is the main idea of the algorithm which follows. We first have some definitions.

Each processor p_i, after its j^{th} event e_{ij}[1] records a triple $\{ps_i, m, M\_sent_{ij}\}$ in volatile storage where ps_i is the state of the processor p_i before the j^{th} event, m is the message (including the identity of the sender which is available as m.SENDER) responsible for the event e_{ij} and M_sent_{ij} is the set of messages (including the destination) sent by p_i in event e_{ij}. From time to time, each processor independently saves the contents of the volatile log in stable storage and clears the volatile log. For an arbitrary processor p_i, let REC_i denote the current recovery point. Let $SENT_{i\rightarrow j}(REC_i)$ represent the total number of messages sent by p_i to p_j and let $RECEIVED_{i\leftarrow j}(REC_i)$ be the total number of messages received by p_i form p_j (from the beginning of the application program) till the recovery point REC_i. We first present an informal description of the recovery algorithm.

The algorithm consists of |V| iterations. The first iteration at a processor starts when it is one of the failed processors that restarts after the failure (called a faulty processor), or it is one of the non-faulty processors and it knows about the failure of another processor[2]. During the beginning of the first iteration, each processor finds the (temporary) recovery point based only on the local information. Processor p_i sets REC_i to the latest event logged in the stable storage if it is a faulty processor and its sets REC_i to the latest event that took place in p_i if it is a non-faulty processor. For each neighbor p_j, p_i computes $SENT_{i\rightarrow j}(REC_i)$ and p_i sends a message $rollback(SENT_{i\rightarrow j}(REC_i))$ to p_j. It now waits for a *rollback* message from each neighbor. This completes the first iteration of the recovery algorithm at p_i. In general, a processor proceeds to the next iteration only after it receives a *rollback* message from every neighbor during the current iteration.

During the k^{th} iteration, processor p_i processes the *rollback* messages it received from all of its neighbors in the $k-1^{st}$ iteration. Let $rollback(c)$ be a message received by processor p_i from its neighbor p_j. Recall that REC_i is the current recovery point for processor p_i. Processor p_i scans its log and determines $RECEIVED_{i\leftarrow j}(REC_i)$, the total number of messages it received from p_j until REC_i. If $RECEIVED_{i\leftarrow j}(REC_i) > c$, it is clear that p_j rolled back to a state such that it (processor p_j) sent only c messages totally to p_i till its (p_j's) current rollback state while REC_i denotes that p_i has received more than c messages. Thus, for the state of p_i to be consistent with respect to the rollback state of p_j, p_i must roll back. P_i examines its log, finds the latest event e such that $RECEIVED_{i\leftarrow j}(e) = c$ and sets REC_i to e. On the other hand if $RECEIVED_{i\leftarrow j}(REC_i) \leq c$, there is no need for p_i to roll back further as its current rollback point is consistent with respect to the current recovery point of p_j. In this manner, all of the *rollback* messages are processed and REC_i is updated. After processing all of the *rollback* messages, p_i determines $SENT_{i\rightarrow j}(REC_i)$ for each neighbor and sends the value in a *rollback* message. This concludes the k^{st} iteration. As explained earlier, p_i starts the $k+1^{st}$ iteration after it receives a *rollback* message from all of its neighbors.

At the end of |V| iterations, the recovery procedure ends and REC_i denotes the recovery point for p_i. A formal description of the algorithm can be found in Figure 3.2. We now present an example before proving that the scheme works correctly.

Example

Consider a distributed computer system consisting of four processors. Figure 3.3 shows a run of the system when an application program is run.

Assume that processors p_2 and p_4 fail and both restart from the most recent checkpointed states c_{21} and c_{41}, respectively. For convenience, let an event e of processor p represent its recovery point, such that the state of p after e is the state used for restarting if e is the current recovery point. In addition, let RP(*) represent a vector of current recovery points and let $RB_i(*)$ represent the vector of *rollback* messages sent by processor p_i during the current iteration where the j^{th} component of $RB_i(*) = SENT_{i\rightarrow j}(REC_i)$. When the recovery procedure is started, i.e., in the first iteration, RP(*) = $[e_{14},e_{24},e_{33},e_{40}]$, $RB_1(*)$ = [-,2,0,0], $RB_2(*)$ = [3,-,1,1], $RB_3(*)$ = [1,2,-,1] and $RB_4(*)$ = [0,0,0,-]. In the second iteration, p_1 processes the $rollback(3)$ message from p_2. Since the value of $RECEIVED_{1\leftarrow 2}(e_{14})$ (i.e. 4) is greater than 3, p_1 needs to roll back to the recovery point e_{13} in response to

1 It is not necessary to save processor states every time; saving processor states only when volatile log is empty is sufficient.
2 Assume that when a failed processor restarts, it broadcasts a message informing other processors about its failure. See [11] for broadcasting a message using O(|E|) messages where |E| is the total number of links.

```
Procedure rollback_recovery
/* procedure executed by processor p_i */
begin
  if p_i is a faulty processor then
    REC_i ← the latest event logged in the stable
     storage;
  else
    REC_i ← the latest event that took place in p_i;
  endif;
  for k ← 1 to |V| do /* there are |V| iterations */
    for each neighboring processor p_j do
        compute SENT_{i→j}(REC_i) and send a
         rollback(SENT_{i→j}(REC_i)) message to p_j;
    end; /*end for*/
    repeat
       wait for a rollback(c) message from each
        neighbor;
       m ← rollback(c) message received;
       put m into processing queue;
    until (a rollback message from each neighbor
     is received)
    while (processing queue ≠ ∅) do begin
        let m=rollback(c) be a message in
         processing queue;
        delete m from rollback processing queue;
        compute the RECEIVED_{i←j}(REC_i) if m
         came from p_j;
        if RECEIVED_{i←j}(REC_i) > c then begin
            find the latest event e such that
             RECEIVED_{i←j}(e) = c;
            REC_i ← e;
         endif;
    end; /* end while */
  end; /* end for loop */
end; /* end procedure */
```

Figure 3.2: Porcedure rollback_recovery

the *rollback*(3) message form p_2. In the same manner, we get RP(*)=[e_{13},e_{24},e_{30},e_{40}], RB_1(*) = [-,2,0,0], RB_2(*) = [2,-,1,1], RB_3(*) = [0,1,-,0] and RB_4(*) = [0,0,0,-]. In the third iteration, RP(*) = [e_{13},e_{22},e_{30},e_{40}], RB_1(*) = [-,2,0,0], RB_2(*) = [1,-,0,1], RB_3(*) = [0,0,-,0] and RB_4(*) = [0,0,0,-]. In the last iteration, RP(*) = [e_{11},e_{22},e_{30},e_{40}]. As a result, the recovery point for each processor is RP(*) = [e_{11},e_{22},e_{30},e_{40}]. It is easy to see that RP(*) is consistent for this example, and the rollback at each processor is minimum.

Correctness

We now prove that our algorithm is correct.

Lemma 3.1: At the end of each iteration of the recovery procedure, at least one processor will roll back to its final recovery point unless the current recovery points are consistent.

Proof:(sketch) To ensure the consistency of processor states, we let processor roll back. It is clear that during the first iteration, the processor that rolls back finds the correct recovery point. At subsequent iterations, it is impossible for a processor to roll back to a state that is inconsistent with respect ot state of one of its neighbors. A more formal proof is by induction on the number of iterations and by contradiction, and will appear in the full paper.□

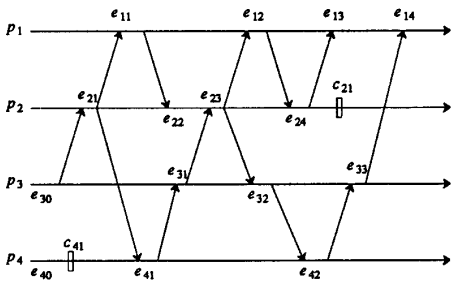

Figure 3.3

Theorem 3.1: At the end of the recovery procedure, all of the processors roll back to the optimum consistent recovery points.

Proof: A processor rolls back to a state only because the state immediatedly following that was started by a message m_1 that transitively depends on an unlogged state of a failed processor. Thus, the rollback points for all processors are optimum. From Lemma 3.1, it is clear that the processor states are consistent.□

We now consider the message complexity of the recovery procedure. Since at least one processor rolls back to its final recovery point at each iteration, it is clear that the number of iterations is at most |V| where |V| is the total number of processors in the system. In each iteration, every processor sends the *rollback* message to each of its neighbors and hence, 2|E| messages are sent in each iteration where |E| is the total number of links in the system. Thus, the total number of messages used is O(|V||E|).

Theorem 3.2: The recovery procedure rolls back processor states to the maximum consistent states using O(|V||E|) messages in the worst case.□

4. RECOVERY WITHOUT ROLLING BACK

In this section, we present another approach to crash recovery -- a technique which does not force any non-faulty processor to roll back. Rolling back processor computations wastes resources and there are numerous situations where recovery with no roll back is desirable. The main idea of the recovery algorithm is to restore the original computation of the failed processors by ensuring that the failed processor receives the same sequence of messages as it did before its failure. For our algorithm to work correctly, we assume that O(1) extra information (one integer value) is added to each application message, and no more than two adjacent processors fail at the same time. A set of processors are said to fail at the same time with regard to the recovery procedure if they fail at the same time or if one fails during the execution of the recovery algorithm initiated because of the failure of another processor. Should more than

two adjacent processors fail, this technique can not be used but the algorithm presented in section 3 can be used to roll back and restart.

From now on we assume that at most two neighboring processors have failed. Note that the total number of failed processors is not restricted to two. For example, in ring networks, two thirds of the processors can fail, and algorithm *No_rollback* is useful as long as there does not exist three consecutive processor failures.

When a processor q fails and restarts from its latest checkpoint, q first checks how many of its neighboring processors failed. If at most one of its neighbors has failed, q starts the recovery procedure No_rollback whose informal description is given below (see Figure 4.2 for a formal description).

When a processor p sends a message m to q during STI i of p, p appends the current STI *index* i to m and sends the message to q. When q receives message m and processes the message, it starts a new STI j and logs the information $\{j, p, m, received\}$ in volatile storage[3]. In the meantime, q also sends back a *received(j)* message to p where j is the STI of q which was triggered by the message sent by p. For each message received by q, it sends a reply (in realty, it is an acknowledgement) to the sender of the message informing the local relative order of this message within processor q with respect to the other messages it received. On receipt of a *received(j)* message from q, p logs the information $\{j, q, m, i, ack\}$ in volatile storage[4].

To recover from the failure, processor q first sends a *failed(j)* message to every neighbor where j is the index of the last STI saved in the stable storage. It then waits for *resend* messages from its neighbors. When processor p receives a *failed(j)* message from q, it (processor p) sends a *resend*($M_{p\to q}(j)$,$\max_p(q)$) message to q. The *resend* message contains two parameters $M_{p\to q}(j)$ and $\max_p(q)$. The first parameter $M_{p\to q}(j)$ represents the sequence of messages sent by p to q that were received and processed by q after STI j of q. Recall that a processor sending a message to another processor receives an acknowledgement from the recipient in the form of a *received* message. The acknowledgement also identifies the STI of the recipient during which the message sent was processed. Thus, p can construct $M_{p\to q}(j)$ easily using the local information. It is easy to see that for processor p, $M_{p\to q}(j) = \{[m_1, index_1],..., [m_k, index_k]\}$, where for $1 \le t \le k$, $\{index_t, q, m_t, *, ack\}$ is in the local log of processor p and $index_t > j$ (here, * denotes a wildcard that matches any number). The sequence of resent messages is a subsequence of the sequence of messages sent by p to q, and is a subsequence of the messages received by q between the latest checkpoint (of q) and the failed point (of q). Processor p has to resend those messages because of q's failure. The second parameter $\max_p(q)$ is the last STI of q during which a message was sent by q to p. In other words, $\max_p(q)$ = STI i of q such that during STI i, q sent a message m to p and STI i is the latest STI. Recall that to each message m that is sent, the sender appends the STI index during which m was generated. Thus, p can get the value of $\max_p(q)$ by examining its local log.

Upon receiving each *resend(M,max)* message, q adds M to its message processing queue and stores max in a local data structure. Recall that each entry of M is of the form $[m, index]$ where message m was sent to q and this message was processed during STI *index* of processor q. After q receives *resend* messages from all of its neighbors, it sorts the messages in the message processing queue in ascending order using the second component (*index*) as the key. Processor q also computes *maxsti*(q) where *maxsti*(q) = maximum $\{ \max_p(q) \mid p$ is a neighbor$\}$. In other words, *maxsti*(q) is the maximum STI index of q that other processors know about. Let the contents of the message processing queue be $\{[m_1,index_1], ..., [m_s,index_s]\}$.

If $index_1 = j+1$ (one more than the index of the last STI in stable storage of q), and $index_1, ..., index_s$ are continuous, it is clear that q has all of the messages and in the correct order. It processes them, and after emptying the message processing queue, it terminates the recovery procedure. On the other hand, if q does not have all of the messages, it is clear that one of its neighbors had failed and q must wait for those messages from the failed processor. In this case, q starts processing messages drawn from the message processing queue as long as the *index*es associated with them are contiguous. When there is a gap, it waits for a message from the failed processor. After the message processing queue is empty and its current STI index is greater than or equal to the *maxsti*(q), q sends a *completed* message to the other failed processor if there is one. If q already received a *completed* message from the neighboring failed processor, q terminates the recovery procedure and begins its normal operation. Otherwise, q waits and processes all of the messages that come from the neighboring failed processor until receives a *completed* message. During execution of the recovery procedure, if q receives any other messages from its neighbors, q adds these messages to a different queue and processes these messages only after completing the recovery procedure.

Example

Consider a distributed computer system consisting of four processors. Figure 4.1 shows the complete run of the system when an application program is run before a failure occurs. In the figure to each message sent the STI *index* (number in <>) is appended, and each sender of a message will receive a *received* message (numbers in ()) from the receiver. For example, message m_1 is appended the STI *index* <2> and p_1 receives a *received* (5) message in return.

[3] It is not necessary to log all of this information and in fact, it is possible to recover from failures even if only {*j,p,received*} is logged.

[4] It is possible to recover from failures even if only {*j,q,i,ack*} is logged.

Assume that two neighboring processors p_2 and p_4 fail. Processor p_2 restarts from the latest checkpoint c_{21} and runs No_rollback procedure. It first sends a *failed*(4) message to all its neighbors. Upon receiving the *failed* message, p_1 sends a *resend* message with m_1 and $\max_1(2)=7$ to p_2. When p_2 finishes processing m_1, it waits for the message m_2 and m_3 from the failed processor p_4. After finishing processing m_3, p_2 knows that it has recovered to STI seven since *maxsti*(2)=7. P_2, then, sends a *completed* message to p_3. Processor p_4 also recovers in a similar manner.

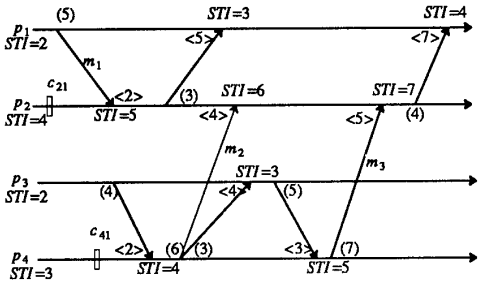

Figure 4.1.

```
/* the recovery procedure is executed by the failed
  processor q after it restarts from its latest
  checkpoint*/
Procedure No_rollback;
begin
  j ← the number of the last STI index in the stable
   storage;
  check how many neighboring failed processors there are;
  if more than two neighboring failed processors then
     run the rollback recovery algorithm and stop;
  else begin
     neighbor_done ← 0;
     send a failed(j) message to every neighbor;
     wait for the resend messages;
     m ← message received;
     repeat
       case m of /* assume m came from p */
          resend message:
             maxsti(q) ← Max(maxsti(q), max_p(q));
             add messages in M_{p→q}(j) to norollback
          processing queue in ascending order;
          application message:
             add m to a different processing queue;
             /* not norollback processing queue*/
          completed message:
             neighbor_done ← 1;
       end /* end case */
     Until (all neighbors send back the recovery
            messages except the other failed processor);
   end; /*end else*/
  endif;
  repeat
    m ← message in the norollback processing queue;
    j ← j + 1;
    if j = index with m then begin
       process message m as a normal application
         message;
       delete m from rollback processing queue;
     end
    else begin
       wait for a resend message from the other failed
        processor;
       m ← message received;
       if m is a resend message then
          process m as a normal application message;
       else /* m is an application message that came
         from a nonfaulty processor*/
          add m to processing queue;
       endif
     end /*end else*/
    endif
  Until (j = maxsti(p))
  send a completed message to the neighboring failed
   processor;
  if neighbor_done ≠ 1 then
   repeat
     wait for the resend message from the failed
      processor;
     /* if the received message is an application
      message, put the message into processing
       queue */
     process the resend message;
   Until (a completed message is received from the
          other failed processor);
end; /* end procedure */
```

Figure 4.2: Procedure No_rollback

Correctness

Recall that we assume the channel is reliable and each time a processor changes its state form s to s', the resulting state s' is determined based only on state s and the contents of the message received. Thus, for a failed processor to execute in a different manner in the second run (after crashing), it must receive a different sequence of messages in the second run which is not possible since the messages that are resent also contain some information about the positions of those messages.

Theorem 4.1: After completing the recovery algorithm, the system can be restored to its original global state.

Proof:(sketch) By ensuring that the sequence of messages a failed processor receives after failure is identical to the sequence of messages received by the same processor before failure, it is easy to see that a failed processor's behavior is the same after failure also. It is easy to verify that the processor states are consistent. A formal proof (by contradiction) is involved and will appear in the full paper.□

Thus, in this scheme, there is no need for nonfaulty processors to roll back and the faulty processors simply re-execute, and all the processor states are restored to consistent and correct states.

We can modify the algorithm to the case when no stable storage is assumed to be present. In this case, non-faulty processors resend all of the messages

they had sent in the original run, and faulty-processors start executing from the beginning instead of restarting from the latest checkpoint. Thus, checkpointing is limited to volatile memory only. Certain optimizations are possible, and the algorithm can be shown to be correct. The details are complex and will appear in the full paper.

5. CONCLUSIONS

The problem of recovering from processor failures is of paramount importance in the design and development of distributed systems. Traditionally, recovery was achieved using checkpointing and rolling back in conjunction with stable and volatile storage. To recover from failures, additional information was added to each message of the application program, thus placing a load on the communication system. In this paper, two new approaches to crash recovery were considered -- one where there is no overhead during normal operation of the system but O(|V||E|) messages are generated in case of processor failure, and another in which no non-faulty processor needs to rollback because of the failure of some of the processors unless more than two adjacent processors fail at the same time. Thus, our techniques are very useful in systems where the possibility of processor failures is low. However, if processor failures are frequent, then the approach of [6] is more desirable since it only uses $O(|V|^2)$ messages in the worst case to recover from the failures of an arbitrary number of processors.

There are several directions in which future work can proceed. Investigating the nature and amount of information to be added to the application messages, establishing lower bounds on message complexities of problems that involve crash recovery, etc. are just some examples, and we are currently working on these and related problems.

References

1. Borg, A., Baumbach, J., and Glazer, S., "A message system supporting fault tolerance," *Proceedings of ACM Symposium on Operating Systems Principles*, pp. 90-99, 1983.
2. Chandy, K.M. and Lamport, L., "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63-75, 1985.
3. Fischer, M.J., Lynch, N.A., and Paterson, M.S., "Impossibility of distributed consensus with one faulty process," *Journal of the Association for Computing Machinery*, vol. 32, no. 2, pp. 374-382, 1985.
4. Gray, J., "Notes on database operating systems: Operating Systems: An advanced course:," *Lecture notes in computer science, 60, Springer-Verlag*, pp. 393-481, 1978.
5. Johnson, D. and Zwaenepoel, W., "Recovery in distributed systems using optimistic message logging and checkpointing," *Proceedings of ACM Symposium on Principles of Distributed Computing*, pp. 171-180, 1988.
6. Juang, T. and Venkatesan, S., "Efficient algorithms for crash recovery in distributed systems," *10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pp. 349-361, 1990.
7. Koo, R. and Toueg, S., "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23-31, 1987.
8. L'Ecuyer, P and Malenfant, J., "Computing Optimal Checkpointing Strategies for Rollback and Recovery Systems," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 491-496, 1988.
9. Leu, P. and Bhargava, B. , "Concurrent Robust Checkpointing and Recovery in Distributed Systems," *Proceedings of the Fourth IEEE International Conference on Data Engineering*, pp. 154-163, 1988.
10. Powell, M. and Presotto, D., "Publishing: a reliable broadcast communication mechanism," *Proceedings of the ninth ACM Symposium on Operating System Principles*, pp. 100-109, 1983.
11. Ramarao, K.V.S. and Venkatesan, S., "Design of distributed algorithms resilient to link failures," *Technical Report, University of Pittsburgh, Pittsburgh*, 1987.
12. Sistla, A.P. and Welch, J., "Efficient distributed recovery using message logging," *Proceedings of Principles of Distributed Computing*, 1989.
13. Son, S.H. and Agrawala, A.K., "Distributed Checkpointing for Globally Consistent States of Databases," *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1157-1167, 1989.
14. Strom, R.E. and Yemini, S., "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 3, pp. 204-226, 1985.
15. Tamir, Y. and Se´quin, C.H., "Error recovery in multicomputers using global checkpoints," *Proc. 13th IEEE Int. Conf. Parallel Processing*, 1984.
16. Venkatesan, S., "Message-optimal incremental snapshots," *Proceedings of the International Conference on Distributed Computing Systems*, pp. 53-60, 1989.
17. Venkatesh, K., Radhakrishnan, T., and Li, H.F., "Optimal Checkpointing and Local Recording for Domino-Free Rollback Recovery," *Information Processing Letters*, vol. 25, no. 5, pp. 295-304, 1987.