Some Thoughts on Distributed Recovery (preliminary version)

Nitin H. Vaidya
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
Phone: 409-845-0512
Fax: 409-847-8578
E-mail: vaidya@cs.tamu.edu

Technical Report 94-044
June 1994

Abstract

This report deals with some aspects of distributed recovery. The report is divided into multiple parts, each part introducing a problem and a solution. The intent of this report is to present a medley of preliminary ideas, more detailed treatment may be presented elsewhere. The report deals with the following problems:

- A single *processor* failure tolerance scheme based on the distributed recovery unit abstraction.
- Staggered checkpointing and coordinated message logging to obtain a consistent logical recovery line.
- Relaxing the definition of *orphan* messages.
- Exploiting network architecture to improve performance of message logging.

1 Introduction

This report deals with some aspects of distributed recovery. The report is divided into multiple parts, each part introducing a problem and a solution. The intent of this report is to present a medley of preliminary ideas, more detailed treatment may be presented elsewhere. Accordingly, the presentation is informal and no correctness proofs are presented. The report deals with the following problems:

- A single *processor* failure tolerance scheme based on the *distributed recovery unit* abstraction. This scheme is derived from a single *process* failure tolerance scheme [7], but tolerates simultaneous failure of all processes on a single processor.

- Staggered checkpointing and coordinated message logging to obtain a consistent logical recovery line. Checkpoint staggering is useful to reduce the *overhead* of a recovery scheme [10]. We present a simple approach that allows staggered checkpoints but bounds the rollback distance by coordinated message logging.

- Relaxing the definition of *orphan* messages. We claim that the commonly used definition of orphan messages can be relaxed. An example is presented to illustrate this.

- Exploiting network architecture to improve performance of message logging. We present a recovery scheme to tolerate a small number of failures in a system based on a wormhole routed network.

The recovery schemes presented here can be viewed as variations of existing recovery schemes, however, we are unaware of any prior work on these variations. A comparison of proposed recovery schemes with existing work is presented in the report.

Part I

2 Single Processor Fault Tolerance Using *DRU*s [12]

The research on message logging schemes for achieving distributed recovery assumes that the system consists of a collection of *recovery units* [3, 11] that are *deterministic*. The execution of such a recovery unit consists of a sequence of piecewise deterministic state intervals, each started by a non-deterministic event. Such an event can be (i) receipt of a message from another recovery unit, (ii) an internal non-deterministic event such as a kernel call, or (iii) creation of the recovery unit [3]. To this we would like to add one more non-deterministic event, namely, delivery of a message sent by the recovery unit to *itself*. The reason will be clearer shortly.

Consider the single fault tolerance schemes proposed by Johnson [7] and Alvisi et al. [1]. Each of these schemes can tolerate failure of a single recovery unit. When multiple processes are scheduled on a single processor, failure of the processor will cause failure of all processes on that processor, not just one. This implies that, for the above schemes to be useful, the collection of processes on a processor should together be considered a *recovery unit*. To differentiate this from traditional recovery unit definition, we name the collection a *distributed recovery unit* (DRU) [12]. This definition of a recovery unit has two related consequences:

- A collection of processes is *not* deterministic, even if each process in the collection is deterministic. The state of the collection depends not only on the messages received from outside the collection, but also on the interleaving of messages between the processes within the collection.
- The DRU sends messages to *itself*. When a process sends a message to another process within the same DRU, it can be interpreted as a message from the DRU to itself. The schemes by Johnson [7] and Alvisi et al. [1] are not designed to handle such messages.

3 Suggested Solution

In this section, we present a single *processor* failure tolerance scheme that treats the collection of all processes on a processor as a *single* distributed recovery unit. This scheme is an extension of Johnson's *sender based message logging* scheme [7]. Similar modifications can also be made to the scheme presented by Alvisi et al. [1].

The recovery scheme presented here is also closely related to Elnozahy and Zwaenepoel [3, 4]. Essentially, each DRU uses Elnozahy' scheme for keeping track of intra-DRU events (using antecedence graph), the single processor fault assumption being used to allow volatile logging of intra-DRU events as well as inter-DRU messages.

In the following, we assume that the system consists of a collection of DRUs, all processes on any given processor forming one DRU.[1] A message sent by a process to another process within the *same* DRU is said to be an intra-DRU message. All other messages are *inter-DRU* messages.

3.1 Failure-Free Operation

Checkpointing: The processes in each DRU periodically take a coordinated checkpoint. The checkpoint of a DRU is *not* coordinated with that of any other DRU. The coordinated checkpoint establishes a consistent recovery line consisting of one checkpoint of each process in the DRU. Along with the checkpoints, the intra-DRU messages that were sent before the recovery line but delivered after the recovery line are also logged on the stable storage.

Sender based message logging: This is similar to Johnson [7] with one important modification. Each DRU maintains an antecedence graph [3] containing *only* the events *internal* to the DRU, including delivery of intra-DRU messages. Specifically, the antecedence graph does not contain events corresponding to inter-DRU messages. For each delivered intra-DRU message a tuple *(sender, receiver, ssn, rsn)* is included in the antecedence graph,

[1] The approach presented here can, in principle, be generalized to the case where the processes within a DRU reside on multiple processors, or different processes on a single processor belong to different DRUs.

where, *sender* and *receiver* are the identifiers of the sender and receiver processes, and *ssn* and *rsn* are the *send sequence number* and *receive sequence number*, respectively. The send and receive sequence numbers are relative to each *process* and not a DRU. That is, each process independently determines SSN and RSN of each message it sends and receives.

This antecedence graph is shared by all processes in the DRU. The antecedence graph is purged each time a message is received by a process in the DRU, as described below.

When an intra-DRU message is delivered, the antecedence graph is updated. The intra-DRU messages need not be logged (except those logged during the checkpointing step described above).

Johnson [7] logs a message and its *receive sequence number* (RSN) in the volatile storage of the sender. We modify his protocol as follows: When an *inter-DRU* message is delivered to a process, it sends the RSN of the message to the message sender. The RSN is tagged by the antecedence graph. After the RSN is sent, the antecedence graph is purged. (The retransmission protocol will retain a copy of the graph and the RSN until they are acknowledged.) Any future internal events will create a new antecedence graph. The message sender logs the antecedence graph in its volatile storage along with the RSN and the message. The steps for logging inter-DRU messages can be summarized as follows:

- Sender S sends an inter-DRU message to receiver R. Sender keeps a copy of the message in its volatile log.
- Receiver determines the RSN of the message and sends the RSN as well as the antecedence graph to sender S.
- Receiver R cannot send any messages until it receives an acknowledgement (of RSN and antecedence graph) from S.

Output Commit Before an output can be committed, the antecedence graph must be logged either in the stable storage or at another processor. (The graph can also be purged at this time, though this will require some modification to the recovery protocol below.)

3.2 Recovery

Assume that a single processor is faulty resulting in the failure of all the processes scheduled on that processor. To recover from the failure, all the processes on the faulty processor are restored to their most recent (coordinated) checkpoint. All the inter-DRU messages received by the processes are sent to the processes in the RSN-order. Along with the inter-DRU messages, their tags (i.e., portions of the antecedence graph) are also sent. These tags indicate which intra-DRU messages must be delivered (and their order) before the inter-DRU message can be delivered. In effect, the antecedence graph indicates the order of execution of the processes within the DRU to be able to recover from the failure. The antecedence graph logged when committing the most recent output may also be needed to recover from the failure, if no inter-DRU message was logged subsequently.

The above recovery procedure is essentially identical to that presented in [3] if the inter-DRU messages are treated as input/output (during recovery only).

4 Relation to Existing Work

The approach presented above is a combination of the recovery schemes presented by Johnson [7] and Elnozahy and Zwaenepoel [4]. [4] presents a recovery scheme that uses coordinated checkpointing as well as logs the antecedence graph. Thus, the above system can be viewed as a collection of DRUs, each DRU using Elnozahy and Zwaenepoel [4] internally (to recover intra-DRU messages) and Johnson and Zwaenepoel [7] externally (to recover inter-DRU messages). Single DRU failure assumption allows the antecedence graph of each DRU to be logged at any other DRU.

Our approach can be viewed as combining two recovery schemes to obtain a *hybrid* recovery scheme [12]. Lowry et al. [9] propose an approach for *hierarchical* implementations, wherein different clusters of recovery units internally use different recovery schemes. The messages between different clusters are sent through interface recovery units that facilitate optimistic message passing between the clusters. Although our approach as well as [9] partition the processes into clusters (or DRUs), the two approaches are complementary to each other (not identical).

5 Summary

The recovery scheme presented above tolerates the failure of a single processor (independent of how many processes are scheduled on the processor). It uses the techniques presented previously in [3, 4, 7, 12] to obtain a useful recovery scheme. An advantage of this scheme is that the antecedence graphs do not become bigger with the size of the system, as only intra-DRU events are included in the antecedence graphs. The hybrid approach can potentially be applied to other recovery schemes as well. It is also possible to extend this approach to relax the requirement that the processes on each processor form one DRU. Specifically, the DRU may contain processes on different processors, also, processes on the same processor may belong to different processors. The issue of dynamically changing the membership of processes to DRUs also needs to be investigated [12].

Part II

6 Completely Staggered Checkpointing

The problem of reducing the overhead of checkpointing in multicomputers has received considerable attention (e.g., [10, 4]). Plank [10] presented different techniques to reduce this overhead. One of the suggested techniques is staggering of checkpoints. When processes in a system coordinate their checkpoints, they tend to take the checkpoints at about the same time. This can result in severe performance degradation in multicomputers where the I/O bandwidth is not adequate. Staggering techniques suggested by Plank alleviate this problem to some extent. However, his algorithm cannot stagger the checkpoints arbitrarily, that is, some checkpoints cannot be staggered without blocking some processes. Plank [10] shows that staggering indeed reduces the overhead significantly for many applications. Here, we present a simple alternative for coordinated checkpointing that allows arbitrary staggering of checkpoints. The solution presented below is closely related to [2, 6, 10, 14], as discussed later.

7 Suggested Solution

The solution suggested here can be summarized as follows:

staggered checkpoints + *coordinated message logging* = *consistent logical checkpoints*

The basic idea is to coordinate *logical* checkpoints [14, 15] rather than *physical* checkpoints. A physical checkpoint of a process is taken by saving the process state on the stable storage. A logical checkpoint is taken by logging all the message received by the process since its most recent physical checkpoint on the stable storage. Thus, a physical checkpoint is trivially a logical checkpoint, however, the converse is not true.

Algorithm

1. A checkpoint coordinator sends a *take_checkpoint* message to each process. Each process, sometime after receiving this message, takes a *physical* checkpoint and sends an acknowledgement to the coordinator. The checkpoints taken by the processes can be

staggered by allowing an appropriate number[2] of processes to take checkpoints at any time (this can be done using a l-mutual exclusion algorithm). The checkpoints taken by the processes need not be consistent. The processes take checkpoints as soon as possible after receiving the *take_checkpoint* message (under the staggering constraint).

After a process takes the checkpoint, it can continue execution without blocking for any other process. A process logs each message delivered to it after its physical checkpoint into a volatile storage buffer. Overflows to this buffer are spilled into the stable storage. (Alternately, the process may asynchronously log these messages to the stable storage even if the buffer is not full.)

2. When the coordinator receives acknowledgement messages from all the processes indicating that they have taken a checkpoint, the coordinator initiates the *consistent message logging* phase of the algorithm. In this phase, any coordinated checkpointing algorithm can be used, for example, Chandy and Lamport [2]. The only difference is that when the original algorithm requires a process to take a physical checkpoint, our processes instead take a logical checkpoint by logging the relevant messages received since the physical checkpoint taken in the previous step.

   When the coordinated checkpointing algorithm is complete, the processes can discontinue logging received messages.

The above algorithm reduces the contention for the stable storage by completely staggering the physical checkpoints. However, contention is now introduced in the second step of the algorithm when the processes coordinate message logging. This contention can be reduced by using the limited staggering techniques proposed in [10]. The proposed scheme will perform well if message volume is relatively small compared to checkpoint sizes. A few variations to the above algorithm are possible:

- If a process received too many messages after the checkpoint in the first step, then it may decide to take a physical checkpoint in step 2 (rather than a logical checkpoint).

[2] For instance, the number of processes allowed to checkpoint simultaneously may be equal to the number of I/O channels.

This makes the physical checkpoint taken by the process in step 1 redundant. However, the overhead may be reduced when checkpoint size is smaller than the message log.

- A process may decide to not take the checkpoint in step 1, if it a priori knows that its message log will be large. In this case, the process would take a physical checkpoint in step 2. [3]

- The coordinator may initiate the *consistent message logging* phase (step 2) even before all processes have taken the physical checkpoint. In this case, consider a process P that receives a "take checkpoint" message (as a part of the coordinated checkpointing algorithm used in step 2) before P has taken the physical checkpoint (required in step 1). Then, process P can take a physical checkpoint in step 2 rather than a logical checkpoint (essentially, process P can pretend that it decided to not take a checkpoint in step 1).

This algorithm establishes a consistent recovery line consisting of one logical checkpoint per process.

Recovery: After a failure, each process rolls back to its recent physical checkpoint and reexecutes (using the logged messages) to restore the process state to the logical checkpoint that is a part of the most recent consistent recovery line.

8 Relation to Existing Work

The algorithm presented above is closely related to [2, 6, 10, 14]. Our algorithm is designed to bound the rollback distance, similar to the traditional coordinated checkpointing algorithms. It may be noted that, after a failure, a process rolls back to a physical checkpoint and then executes to restore a logical checkpoint. Thus, the overhead of recovery (or rollback distance) is determined by when physical checkpoints are taken.

[3] Johnson [6] suggested a scheme where each process uses a similar heuristic to decide whether to log messages or not.

Johnson [6] presents an algorithm that forces the processes to log message on the stable storage or to take a physical checkpoint. The goal of his algorithm is to make the state of a *single* process committable (primarily, to allow it to commit an output). Also, his algorithm does not control the time at which each process takes the checkpoint. Our algorithm is designed to bound the rollback distance (and not for output commits) and it makes recent states of *all* processes committable. The same result can be achieved by executing Johnson's algorithm *simultaneously* for *all* processes. The implementation will not bound the rollback distance, however, as the timing of the physical checkpoints is not controlled by his algorithm. Additionally, Johnson's algorithm can result in all messages being logged (as processes may log messages asynchronously), our algorithm logs messages only until the *consistent message logging* phase is completed.

Plank [10] presents two coordinated checkpointing algorithms (based on Chandy and Lamport [2]) that attempt to stagger the checkpoints. However, it is possible that some checkpoints taken by these algorithms cannot be staggered. In contrast, our algorithm allows arbitrary staggering of the checkpoints.

Wang et al. [14, 15] introduce the notion of a logical checkpoint. [14, 15] determines a recovery line consisting of consistent logical checkpoints, *after* a failure occurs. This recovery line is used to recover from the failure. Their goal is to determine the "latest" consistent recovery line using the information saved on the stable storage. During failure-free operation each process is allowed to independently take checkpoints and log messages. On the other hand, our scheme *coordinates* logical checkpoints *before* a failure occurs. These logical checkpoints are used to recover from a *future* failure. One consequence of this is that we do not need to log all messages, only those message are logged which make the logical checkpoints consistent.

Part III

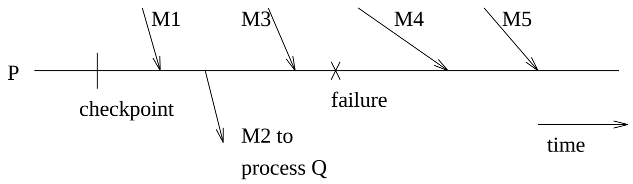

Figure 1: Process P

9 Relaxing the Definition of Orphan Messages

To our knowledge, all message logging protocols require that the *receive sequence number* [5] of a message must be logged before the message is considered to be fully logged. Our intent here is to show that this is not a *necessary* condition to be able to achieve recovery, but a performance optimization.

To illustrate this point, consider a single *deterministic* process P shown in Figure 1. The state of process P is completely defined by its initial state and the messages it receives. Process P takes a checkpoint and then receives messages M1 and M3. Process P sends a message M2 to some process Q after receiving M1. Messages M4 and M5 have been sent to process P but have not yet been received by P. Process P fails after receiving M3. Assume that the receive sequence numbers of messages M1 and M3 are unknown to anyone other than P. The traditional optimistic message logging protocols would declare that message M2 is an orphan, as the state in which process P sent the message cannot be recovered. This would cause rollback propagation to process Q.

We claim that message M2 is not really an orphan massage. Let it be known that no messages other than those shown in the figure have been sent to process P since its recent checkpoint. Assume that messages M1, M2, M3, M4 and M5 are available in spite of the failure of process P. The RSNs of messages M1, M3, M4 and M5 are not known, however.

The goal of our recovery protocol is to recreate the state in which message M2 was sent, as this would avoid rollback propagation. It is known that four messages were sent to process P since its last checkpoint, though the order of delivery (or for that matter, which message were delivered before failure) is not known. It is possible to try all 16 (factorial of 4) permutations in which the four messages could have been delivered since the latest checkpoint. The first permutation that results in message M2 being resent can be assumed to be correct. It is clear that at least one permutation will result in M2 being sent, and the state of process P can be restored to the state in which M2 was sent. Process P can receive any remaining messages subsequently. This procedure ensures that the rollback is not propagated to process Q (even though M2 is an orphan message by traditional definitions). If multiple messages were sent by P before failure, then *all* these messages must be reproduced by one of the permutations. This approach can potentially be generalized to tolerate multiple failures.

Time complexity of the method presented above increases substantially with (i) the number n of processes that have sent a message to P whose RSN is not logged, and (ii) number f of faulty processes. This may make the above approach impractical, as recovery could take an inordinately long time.

The advantage of the above approach is that message M2 could be committed (if it were an output) even before it is known that M1 is fully logged (i.e., message and RSN both).

The above approach may be useful with some constraints. For instance, one can bound the number of received messages that are not fully logged. For example, let us assume that the state of a process can be assumed to be recoverable only if at most one received messages is not fully logged. This would reduce the time complexity of recovery, as the number of permutations to be tried is linear in n (rather than factorial).

The objective here was to demonstrate that the commonly used definitions of orphan messages impose more constraints than absolutely necessary. However, removing these constraints can result in large overheads during failure recovery. Whether relaxation of the constraints is useful in practice needs to be investigated further.

Part IV

10 Exploiting Hardware to Improve Performance

The following presents a simple recovery scheme that tolerates k faults (for some $k \geq 1$). This scheme is based on single fault tolerance schemes presented by Alvisi et al. [1] and Johnson and Zwaenepoel [7]. Although the proposed scheme may seem very expensive, for $k \leq 3$, an efficient implementation of this scheme on wormhole-routed networks is possible. The implementation of the recovery scheme is sketched subsequently. To simplify our discussion assume that only one process is scheduled on each processor. The terms *process*, *processor* and *node* are used interchangeably.

The recovery scheme allows each process to checkpoint independently. To ensure that recovery is possible in spite of k simultaneous failures, each message is logged on k nodes, excluding the receiver. The message logging protocol is described below. Note that the implementation of this protocol will exploit the wormhole routing network, and will eliminate many messages required in the protocol below.

- When a process S sends a message M to a process R, message <S,SSN,M> is also sent to $k-1$ other processes, where SSN is the send sequence number of the message. The message is logged by the sender S and these $k-1$ processes in their respective volatile storages. This way, there will be k copies of <S,SSN,M>. Sender S should receive an acknowledgement from the $k-1$ nodes on receiving the message, to ensure that the message is logged. When process S receives all the acknowledgments, it sends a message to R informing that it has received these acknowledgements.

- On receiving message M, the receiver process R sends a message containing <S,SSN,RSN,R> to k nodes, to be logged in their volatile storages. These nodes may or may not include the nodes at which <S,SSN,M> is logged.

- A process on receiving <S,SSN,RSN,R>, sends an acknowledgement to process R. Process R cannot send any message until (i) it has received acknowledgements from k such processes, and (ii) it knows that <S,SSN,M> has been logged at k nodes.

Recovery: Consider a situation where receiver R fails and the total number of faulty processes is at most k. As there are k copies each of <S,SSN,M> and <S,SSN,RSN,R> (at k nodes other than the receiver), both will be available in spite of the failures.

When a failure occurs, a process rolls back to its most recent checkpoint and requests other nodes to send <S,SSN,M> and <S,SSN,RSN,R>, so that it can recover to its most recent state. It is easy to see that this scheme can tolerate k simultaneous failures. After the faulty processes have recovered from the failure, all the processes take a coordinated checkpoint. Recovery is considered to be completed when this checkpoint has been taken. The reason for taking the coordinated checkpoint is that the faulty processes loose some of the log information that is not recovered during recovery. To protect the system from future failures, it is useful to take a coordinated checkpoint. Such a checkpoint is taken only when a failure occurs. (It is also possible design a protocol to reconstruct the logs after a failure. However, we believe that taking a coordinated checkpoint is a simpler alternative.)

11 Implementation on a Wormhole-Routed Network

The wormhole routed networks are characterized by "pipelined" movement of messages through the network. A message is typically routed through intermediate nodes before it reaches the destination. This provides an opportunity for the intermediate nodes to "snoop" on the message (or make a copy of the message) when the message is routed to the next node on the path to the destination. Also, the length of the path has a small effect on the communication latency. These properties of wormhole routed networks are exploited in the proposed implementation of the k-fault tolerance algorithm.

It is possible to conceive implementation of the proposed algorithm using various wormhole routing protocols. The discussion in this report, however, utilizes a variant of the the *fault-tolerant compressionless routing* (FCR) recently proposed by Kim et al. [8]. The next section briefly describes FCR. The subsequent sections describe the routing protocol used by our algorithm, and the proposed implementation of the k-fault tolerance algorithm.

11.1 Fault-Tolerant Compressionless Routing [8]

FCR tolerates failures in routing, i.e., a message is delivered to its destination in spite of few link (or router) failures. FCR is not designed to recover from processor failures, however.

FCR is used to send messages between the nodes. The basic idea behind FCR is to pad, if necessary, a message with "adequate" number of empty flits. This padding is used to provide the sender an implicit acknowledgement of message receipt by the receiver. Thus, an (implicit) acknowledgement is received for every message sent by the FCR protocol. Proposed implementation of the k-fault tolerance algorithm uses an *L-snooping protocol* developed with FCR as the starting point.

11.2 L-Snooping Protocol Based on FCR

The basic features of the L-snooping protocol are listed below:

1. Each message sent by a sender S to receiver R is routed a path consisting of at least $L+1$ processors, including S and R. (This may require some modifications to the routing algorithm as discussed later.) The value of L is specified by the sender.

2. Similar to FCR, an *adequate* number of *padding* flits are added to each message to ensure that the sender S receives an implicit acknowledgement of the message.

3. The first L processors on the path, including sender S, make a copy of the message being transmitted, and store it in a local log on the node. This is achieved by including a counter with the message. The counter is initialized to L by the sender. Each subsequent node makes a copy of the message if the counter is non-zero. If the counter[4] is non-zero, the node also decrements it by one before passing on to the next node on the path to the destination. As long as the counter can be stored in a single flit, the counter should not cause significant performance degradation.

[4] An alternative is to use a "counter" that is initialized to a string of 1s followed by 0s (e.g., 11000). Each node makes a copy of the message if the counter is non-zero and resets the least significant 1 in the counter.

The implicit acknowledgement provided by FCR eliminates the need for sending explicit acknowledgements required in Steps 1 and 3 of the k-fault tolerant algorithm in Section 10.

Implementation of the L-snooping protocol depends on the choice of L, as each message must travel on a path with at least $L+1$ nodes. Additionally, the routing algorithm must be deadlock-free.

To tolerate k failures, a message needs to be logged at $k-1$ nodes other than the sender. Thus, the L-snooping protocol with $L = k$ will be useful. As we are interested in small values of k, in this report, we consider protocols that guarantees that each message will travel to least $k+1$ nodes, provided $k \leq 3$. $L \leq 3$ is adequate when $k \leq 3$.

Deliberate Misrouting

When the number of nodes on the shortest path between the sender and the receiver is $k+1$ or more, no special treatment is required. When the number of nodes on the shortest path is smaller than $k+1$, the message is deliberately misrouted to ensure that it will traverse at least $k+1$ nodes. As we are assuming that $k \leq 3$, only messages between nodes at distance 1 and 2 need to be misrouted.[5]

For m-ary-n-cube, the rules for routing a message from source S to receiver R are as follows, assuming that $k \leq 3$:

- If the distance between S and R is less than k, then the source S misroutes the message to a *neighbor* that is *not* on a shortest path to node R.

- Each node, other than the source, tries to route the packet to the destination using FCR with an additional constraint: if node A routed the packet to node B, then node B does not route the packet back to node A.

The above rules imply that if the distance between a pair of nodes is $D < k$, then the packet will travel a path containing at least $D+3$ nodes (i.e., $D+1$ nodes on the shortest path

[5] Paths between nodes at distance 3 and more contain at least 4 nodes, including the sender and the receiver.

+ two more nodes due to the misrouting at the source). As $D \geq 1$, this implies that each packet will travel to at least 4 nodes, which satisfies our requirements when $k \leq 3$.

We now describe our protocol for $k \leq 3$ in detail. This protocol can potentially be generalized to arbitrary k. Generalization needs design of deadlock-free routing protocols that guarantee paths of length at least $k+1$. There is another alternative for logging messages at k nodes. The alternative requires that a message be sent on multiple paths (to different destinations) such that at least $k+1$ nodes are traversed by these paths.

11.3 k-Fault Tolerance

This section describes how the k-fault tolerance algorithm in Section 10 be implemented on a wormhole routed network. As stated previously, the scheme allows each process to checkpoint independently. To ensure that recovery is possible in spite of k simultaneous failures, each message is logged on k nodes (excluding the receiver), using the message logging protocol described below:

1. When a process S sends a message M to a process R, the SSN of the message and identifier of S are included with the message. The message <S,SSN,M> is sent using k-snooping. Therefore, the sender receives an implicit acknowledgement that the message is received by R. When R receives the entire message, it can correctly assume that the message is logged by k nodes.

2. On receiving the message, R determines the receive sequence number of the message, and sends a message containing <S,SSN,RSN,R> to any node v using $(k+1)$-snooping. The choice of node v is arbitrary. $(k+1)$-snooping guarantees that the message will be logged at k nodes other than R. Some heuristic may be used to choose node v.

   An optimization to reduce the overhead of this step is possible, as described later.

   A mechanism is necessary to distinguish between the two types of messages, namely, <S,SSN,M> and <S,SSN,RSN,R>. A "type" field can be used to distinguish between the two types.

3. No other message can be sent by R until it receives an acknowledgement for the message sent in the above step.

The recovery protocol is presented in Section 11.4.

The above algorithm ensures that <S,SSN,RSN,R> is logged at k nodes (excluding R) before R sends any message. However, the correctness of the algorithm only requires that <S,SSN,RSN,R> be logged at k nodes (excluding R) before a subsequent message from R is *received* by any process. To satisfy this condition, the <S,SSN,RSN,R> message need not be sent as a separate message. It can be tagged to any other message that process R wants to send after receiving message M from process S. However, process R should not send another message until an (implicit) acknowledgement is received for this message. Another alternative is to allow the process to send a message before the acknowledgement is received, but tag the RSN information to these messages as well. This is similar to the single fault tolerance scheme presented by Alvisi et al. [1].

11.4 Recovery Protocol

Each faulty process can recover independently. When a failure occurs, a faulty process P rolls back to its most recently saved checkpoint. To recover the state prior to failure, process P should receive the same messages as it received before the failure (in addition, order of message receipt should also be retained).

The checkpoint of P contains RSN of the last message received before the checkpoint was taken. Therefore, P knows the RSN of the first message it received after the checkpoint, say *rsn1*. During recovery, P broadcasts a request requesting the SSN of the message corresponding to *rsn1*. A node that contains in its log an entry of the form <Q,*rsn1*, *ssn1*,P>, responds by sending the entry to P. (There are potentially k such nodes). When P receives the response, it knows that the message was sent by Q. Process P then sends a request to node Q requesting message with RSN = *rsn1*. If the request could not be sent to Q because of some failure, then P broadcasts a request asking for the *rsn1*-th message of process Q. At least one fault free node is guaranteed to respond to this request, as k nodes (excluding P) must have a copy of the message, of which at most $k-1$ can be faulty. Similarly, process P can obtain all the subsequent messages. Eventually, process P will receive all the messages it had received before failure. The next request for the SSN of a message will not be replied to by any node, indicating that process P has recovered from failure. After receiving fully logged messages, P can receive the partially logged messages in any order. A message M is partially logged if <sender id, SSN, M> is logged but the RSN of the message is not logged.

After the faulty processes have restored their state, the system takes a coordinated checkpoint. The system is considered to have recovered from failures when this checkpoint is completed.

12 Summary

Part IV of the report presented a recovery scheme targeted for multicomputer systems based on wormhole-routed networks. This scheme exploits wormhole routing to minimize the number of messages. It is designed to tolerate a small number of simultaneous failures. (Motivation for design of such schemes is presented in [13].) The work presented here demonstrates that (i) there is potential for design of efficient recovery schemes for small number of failures, and (ii) it is possible to exploit hardware to improve performance of such schemes.

References

[1] L. Alvisi, B. Hoppe, and K. Marzullo, "Nonblocking and orphan-free message logging protocols," in *Digest of papers: The* 23^{rd} *Int. Symp. Fault-Tolerant Comp.*, pp. 145–154, 1993.

[2] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states in distributed systems," *ACM Trans. Comp. Syst.*, vol. 3, pp. 63–75, February 1985.

[3] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit," *IEEE Trans. Computers*, vol. 41, May 1992.

[4] E. N. Elnozahy and W. Zwaenepoel, "On the use and implementation of message logging," in *Digest of papers: The* 24^{th} *Int. Symp. Fault-Tolerant Comp.*, pp. 298–307, June 1994.

[5] D. B. Johnson, *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Computer Science, Rice University, December 1989.

[6] D. B. Johnson, "Efficient transparent optimistic rollback recovery for distributed application programs," in *Symposium on Reliable Distributed Systems*, pp. 86–95, October 1993.

[7] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging," in *Digest of papers: The* 17^{th} *Int. Symp. Fault-Tolerant Comp.*, pp. 14–19, June 1987.

[8] J. H. Kim, Z. Liu, and A. A. Chien, "Compressionless routing: A framework for adaptive and fault-tolerant routing," in *Int. Symp. Comp. Arch.*, pp. 289–300, 1994.

[9] A. Lowry, J. R. Russell, and A. P. Goldberg, "Optimistic failure recovery for very large networks," in *Symposium on Reliable Distributed Systems*, pp. 66–75, 1991.

[10] J. S. Plank, *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Dept. of Computer Science, Princeton University, June 1993.

[11] R. E. Strom and S. A. Yemini, "Optimistic recovery: An asynchronous approach to fault-tolerance in distributed systems," *Digest of papers: The* 14^{th} *Int. Symp. Fault-Tolerant Comp.*, pp. 374–379, 1984.

[12] N. H. Vaidya, "Distributed recovery units: An approach for hybrid and adaptive distributed recovery," Tech. Rep. 93-052, Computer Science Department, Texas A&M University, College Station, November 1993.

[13] N. H. Vaidya, "A case for multi-level distributed recovery schemes," Tech. Rep. 94-043, Computer Science Department, Texas A&M University, College Station, May 1994.

[14] Y. Wang, Y. Huang, and W. K. Fuchs, "Progressive retry for software error recovery in distributed systems," in *Digest of papers: The* 23^{rd} *Int. Symp. Fault-Tolerant Comp.*, pp. 138–144, 1993.

[15] Y. M. Wang, A. Lowry, and W. K. Fuchs, "Consistent global checkpoints based on direct dependency tracking." To appear in *Inform. Process. Lett.*