# A Case for Two-Level Distributed Recovery Schemes

Nitin H. Vaidya
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112, U.S.A.
E-mail: vaidya@cs.tamu.edu

## Abstract

Most distributed and multiprocessor recovery schemes proposed in the literature are designed to tolerate arbitrary number of failures. In this paper, we demonstrate that, it is often advantageous to use "two-level" recovery schemes. A *two-level* recovery scheme tolerates the *more probable* failures with low performance overhead, while the less probable failures may be tolerated with a higher overhead. By minimizing the overhead for the more frequently occurring failure scenarios, our approach is expected to achieve lower performance overhead (on average) as compared to existing recovery schemes.

To demonstrate the advantages of two-level recovery, we evaluate the performance of a recovery scheme that takes two different types of checkpoints, namely, 1-checkpoints and $N$-checkpoints. A single failure can be tolerated by rolling the system back to a 1-checkpoint, while multiple failure recovery is possible by rolling back to an $N$-checkpoint. For such a system, we demonstrate that to minimize the average overhead, it is often necessary to take *both* 1-checkpoints and $N$-checkpoints.

While the conclusions of this paper are intuitive, the work on design of appropriate recovery schemes is lacking. The objective of this paper is to motivate research into recovery schemes that can provide multiple levels of fault tolerance.

## 1 Introduction

Many applications require massive parallelism to solve a problem in a reasonable amount of time. Such applications encounter a high failure rate due to large multiplicity of hardware components. In the absence of a failure recovery scheme, the task must be restarted (from beginning) whenever a failure occurs. This leads to unacceptable performance overhead for long-running applications. Some failure recovery scheme must be used to minimize the performance overhead. *Performance overhead* of a recovery scheme is the increase in the execution time of the task when using the recovery scheme. The *performance overhead* of a recovery scheme consists of two components:

- Overhead during failure-free operation (*failure-free overhead*), e.g., checkpointing and message logging.
- Overhead during recovery (*recovery overhead*).

The objective of this paper is to analyze an approach to reduce the average performance overhead.

The design principle *"make the common case fast"* has been successfully used in designing many components of a computer system (e.g., cache memory, RISC [12]), and some aspects of checkpointing and rollback [14, 21]. However, designers of distributed rollback recovery schemes have largely ignored this guideline. In any system, some failure scenarios have a greater probability of occurring as compared to other scenarios. In the context of failure recovery, the *"common case"* consists of the *more probable* failure scenarios. The above guideline suggests that a recovery scheme should provide low-overhead protection against *more probable* failures, providing protection against other failures with, possibly, higher overhead. We refer to recovery schemes having this capability as *two-level* schemes. This approach can be generalized to *multi-level* recovery [18]. It was recently brought to our attention [3] that, for transaction-oriented systems, Gelenbe [8] previously proposed an approach similar to the *multi-level* recovery approach. Gelenbe's work is discussed in Section 5.

Most existing recovery schemes are *"one-level"* in the sense that their actions during *failure-free* execution are designed to tolerate the worst case failure scenario. For example, the traditional implementations of consistent checkpointing algorithms are designed to tolerate simultaneous failure of all components in the system [13]. The two-level recovery approach can achieve lower overhead than one-level schemes by differentiating between the more probable failures and the less probable failures. In this paper, we analyze a two-level recovery scheme and demonstrate that it can perform better than a one-level recovery scheme. Although a large number of researchers have analyzed checkpointing and recovery (e.g., see [4, 6, 11, 15, 19]), to our knowledge, except for [8], no analysis of two-level recovery schemes has been attempted so far.

Paper organization: Section 2 describes the proposed two-level recovery scheme. Performance analysis is presented in Sections 3 and 4. Related work is discussed in Section 5. The paper concludes with Section 6. The appendix briefly presents an alternate approach to analyze the two-level scheme.

## 2 A Two-Level Recovery Scheme

The proposed recovery scheme is useful for a network of $N$ processors. Each processor has a local volatile memory storage. The processors share a stable storage that can be accessed over the network. To simplify the discussion, each processor is assumed to execute one process.

We consider only crash (fail-stop) failures. Each processor is subject to failures; the occurrence of a processor failure is governed by a Poisson process with failure rate $\lambda$. Failures of the processors are independent of each other. Failure of a processor results in the loss of its volatile storage. The *stable* storage is assumed to be always failure-free.

In the environment under consideration, small number of failures are more probable than a large number of failures. Specifically, *single processor failures* are more probable than all other failure scenarios. The two-level recovery scheme analyzed in this paper consists of two *components*, one *component* recovery scheme designed for single failure tolerance, and the second component scheme designed for tolerating all other failure scenarios. The two component recovery schemes are summarized here:

• **The first component** is the single process failure tolerance scheme presented in [1]. In this scheme, the processes periodically take checkpoints (which need not be consistent with each other). The checkpoint of a process can be saved in any volatile storage except that of its own processor. The communication messages are saved by their senders in their volatile storage. As the messages are simply retained in the volatile storage of their senders, we assume that they do not affect the overhead of the recovery scheme significantly. The failure-free overhead is dominated by the overhead of taking checkpoints.

To simplify analysis, we assume that, the processes take checkpoints at about the same time. Our analysis assumes that the synchronization overhead is included in the checkpoint overhead.

This *component* scheme is capable of tolerating only a single failure. To tolerate a single failure, the faulty process is rolled back to its previous checkpoint (which is saved on a non-faulty processor). Subsequently, the messages that the faulty process had received before failure are re-sent to recover its state. These messages are available in the volatile memory of the message senders.

If a second failure occurs before the system has recovered from the first failure, it is possible that the system may not be able to recover from the failure. We make the pessimistic assumption that this component scheme can never recover from more than one failure. Thus, when multiple simultaneous failures occur the system must be rolled back to the start of the task (or to a consistent state saved on the stable storage, as discussed below). (Two failures are said to be *simultaneous* if second failure occurs before system has recovered from the first failure.)

We make a second pessimistic assumption that, when a single processor failure occurs, during recovery, the non-faulty processors do not perform any useful computation. In other words, we assume that the non-faulty processes block until the faulty processor has recovered. This may not always be true for all applications. In spite of the pessimistic assumptions, we show that the two-level recovery scheme can perform better than a traditional one-level recovery scheme.

We refer to the checkpoints taken by this *component* scheme as *1-checkpoints*, as they are useful to recover from single failures only. A *checkpoint interval* is the duration between two adjacent checkpoints. For this scheme, the failure-free overhead per checkpoint interval is denoted by $C_1$. $C_1$ is the increase in the execution time of a checkpoint interval due to the use of this recovery scheme. As noted earlier, $C_1$ is assumed to be dominated by the overhead of taking 1-checkpoints.

• **The second component** recovery scheme periodically saves *consistent*[1] global checkpoints on the stable storage. To establish the checkpoint, the processes coordinate with each other and ensure that their states saved on the *stable* storage are consistent with each other. Such a checkpoint is useful to recover from an arbitrary number of failures. Therefore, these checkpoints are called *N-checkpoints*. For this *component* scheme, the failure-free overhead per checkpoint interval is denoted by $C_N$. $C_N$ includes the overhead of checkpoint coordination. Volatile storage access is often cheaper than accessing the shared stable storage. Therefore, we expect that $C_1 < C_N$.

The two-level recovery scheme analyzed here consists of the above two components. This two-level scheme takes 1-checkpoints more frequently and $N$-checkpoints less frequently. As the 1-checkpoints are taken more frequently, recovery overhead for a single processor failure is smaller. Also, overhead of taking 1-checkpoints is lower than that of $N$-checkpoints. As will be demonstrated in this paper, the two-level scheme can achieve better performance as compared to either component recovery scheme.

To further clarify the concept of *two-level* recovery, the tables below present an analogy of the two-level recovery scheme with cache memory organizations.

Cache and main memory (two-level) hierarchy

| access type | served by | access time |
|---|---|---|
| address in cache | cache | small |
| address not in cache | main mem. | large |
| average access time = small | | |

Two-level recovery scheme

| failure scenario | failure tolerated by | overhead |
|---|---|---|
| single failure | 1st component scheme | small |
| other | 2nd component scheme | large |
| average performance overhead = small | | |

We assume that the processes take equi-distant checkpoints, adjacent checkpoints being separated by $T$ time units. Every $k$-th checkpoint is an $N$-checkpoint ($k \geq 1$) and all others are 1-checkpoints. Thus, the interval between two consecutive 1-checkpoints is $T$ and the interval between two consecutive $N$-checkpoints is $kT$ (excluding the time required to take 1-checkpoints). Figure 1 illustrates this for $k = 3$. (Empty boxes represent a 1-checkpoint, while shaded boxes represent $N$-checkpoints.) Figures in this paper illustrate execution of the task using a single horizontal line, as in Figure 1. The task consists of $N$ processes, however, as their checkpoints occur at about the same time, the checkpoints are shown on the execution line using a single box.

The execution time (length) of the task, in a failure-free environment (without using any recovery scheme), is denoted by $\Upsilon$. It is assumed that $\Upsilon$ is an integral multiple of $T$, say $\mu T$ where $\mu$ is a positive integer. However, $\Upsilon$ is *not* necessarily an integral multiple of $kT$.

The interval between any two consecutive checkpoints is called a *1-interval*. The execution of the task is divided into certain number of *segments*, each segment terminating with an $N$-checkpoint. For example, in Figure 1, the task is divided into four segments.

We assume that no checkpoint needs to be taken at the

[1]A *consistent* global checkpoint consists of one checkpoint per process such that a message sent after the checkpoint of one process is not received by another process before taking its checkpoint [5].

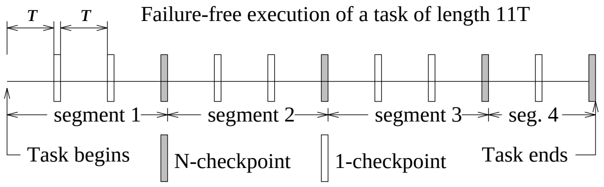


Figure 1: 1-checkpoints and $N$-checkpoints

beginning of the task, and an $N$-checkpoint is taken at the completion of the task. This implies that, when $\Upsilon$ is not an integral multiple of $kT$, the last segment is shorter than the rest. For example, in Figure 1, length of the task is $11T$ and $k = 3$. Therefore, computation time in the last segment is $2T$, while that in other segments is $3T$ each.

**Rollback Recovery:** The time required to perform a rollback to a previous checkpoint is assumed to be $R$. (This does not include the time required for re-execution.) Consider a failure that can be tolerated by rolling back to a certain checkpoint CP. If the failure is detected when $t$ time units of computation was performed after checkpoint CP, then it is assumed that $t$ units of execution is required to re-do the lost computation (in absence of further failures). In the past, some researchers have assumed (e.g., [4]) that the time required to re-do the computation is $\alpha t$ for some constant $\alpha$. Thus, we assume $\alpha = 1$ here. However, our analysis can be easily revised when $\alpha \neq 1$.

If at most one failure occurs during the execution of a *1-interval*, the failure can be tolerated by rolling back to the most recent checkpoint. (The most recent checkpoint may be a 1-checkpoint or an $N$-checkpoint.) If, however, a failure also occurs during the re-execution of the same 1-interval, system is rolled back to the most recent $N$-checkpoint (or to the start of the task, if no $N$-checkpoint is taken before the failure).

Figure 2(a) illustrates a scenario where a failure occurs during 1-interval $I_2$, and the system is rolled back to the most recent checkpoint ($CP1$). No failure occurs during the re-execution of $I_2$.

Figure 2(b) illustrates a scenario where a failure occurs during 1-interval $I_2$, and the system is rolled back to the previous checkpoint ($CP1$). Another failure occurs during the re-execution of $I_2$. Therefore, the system is rolled back to the most recent $N$-checkpoint (CP0).

Figure 2(c) illustrates a scenario similar to 2(a). In this case also a failure occurs during interval $I_2$ and no failure occurs during the re-execution of $I_2$. A failure occurring during interval $I_3$ is treated identical to the first failure during $I_2$. That is, the system rolls back to the most recent checkpoint $CP2$. Essentially, failures occurring during two different 1-intervals are treated independently.

## 3 Performance Analysis

The metric of interest here is the *average performance overhead* of the recovery scheme. Let $E(\Gamma)$ denote the expected (average) time required to complete the task using the given recovery scheme. The average overhead is evaluated as a fraction of task length $\Upsilon$. Specifically,

$$\text{average performance overhead} = \frac{E(\Gamma)}{\Upsilon} - 1$$

Average *percentage* overhead is obtained by multiplying the average overhead by 100.

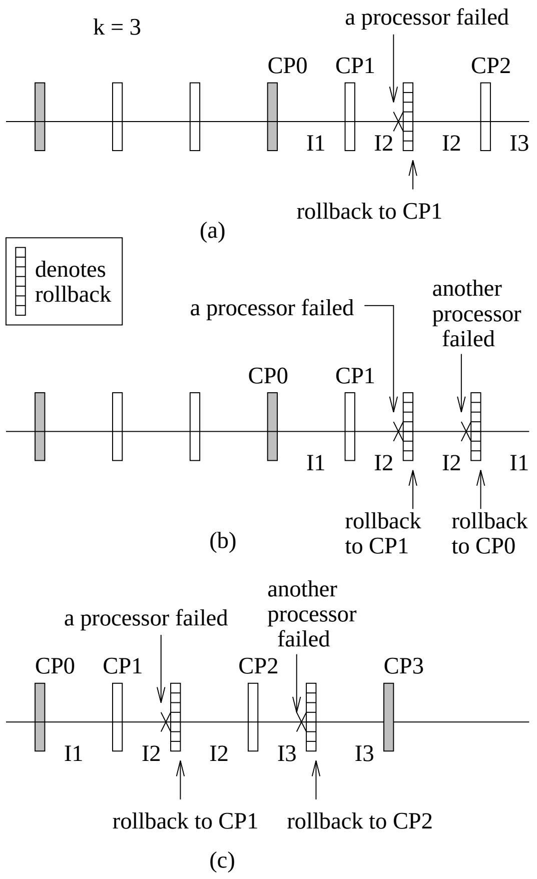


Figure 2: Illustration of fault effects

This section presents an analysis of the average performance overhead. The analysis can be made more intuitive by using Markov chains. The Appendix briefly describes the Markov chain for the proposed scheme. For an example of detailed analysis using Markov chains, the reader is referred to a recent report that presents analysis of *another* two-level scheme [17]. (The following analysis was included in the reviewed manuscript, and therefore, is retained in this publication without significant changes.)

### 3.1 Notation Convention

Two superscripts are used in our notation, namely, $*$ and @. While the exact implications of the superscripts will be clearer as various notation is introduced, the two superscripts are intended to be used as defined below:

- A superscript $*$ denotes that the quantity is related to a 1-interval that terminates with an $N$-checkpoint. Absence of the superscript $*$ generally implies (not always) that the quantity is related to a 1-interval that terminates with a 1-checkpoint.
- A superscript @ denotes that the quantity is related to execution of a segment or a 1-interval that is *not* initiated immediately following a failure. Absence of the superscript @ generally implies (not always) that the quantity is related to execution of a segment or a 1-interval that is initiated immediately following a failure.

## 3.2 Preliminaries

In the following, we use the terms *cost* and *overhead* interchangeably.

Recall that each $N$-checkpoint terminates a *segment* of the task's execution. From the discussion above it is clear that multiple failures cause a rollback to the beginning of the segment during which the failures occur. Additionally, failures while executing one segment do not affect the time required to execute other segments. Therefore, the expected time required to complete the task can be obtained as the sum of expected time required to complete each segment of the task.

For a given $k$, the task is divided into $\lceil \frac{\mu}{k} \rceil$ segments. Each segment, possibly except the first segment, includes a total of $k$ checkpoint (of which $k-1$ are 1-checkpoints). The first segment may contain less than $(k-1)$ 1-checkpoints, as the task length $\Upsilon$ may not be an integral multiple of $kT$. We first evaluate the expected time required to complete a single segment that includes $c$ 1-checkpoints and one $N$-checkpoint, as shown in Figure 3. The $c$ 1-checkpoints are labeled $CP_1$ through $CP_c$, and the $N$-checkpoint at the end of the segment is labeled $CP_{c+1}$. (The analysis below assumes that $c > 0$. The results for the case of $c = 0$ can be obtained similarly.) Observe that the segment consists of $(c+1)$ 1-intervals. Failures may occur while executing any of these intervals. If multiple simultaneous failures occur while executing any one 1-interval, then the system must be rolled back to the start of the segment.

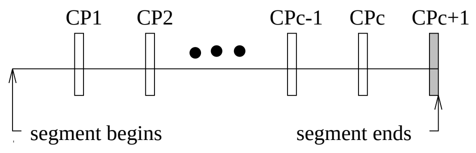


Figure 3: A segment: failure-free execution

We now introduce some notation. (The Appendix relates some of the notation to the Markov chain presented in the appendix.) To understand the notation, recall that a rollback may occur to the start of segment if multiple failures occur while executing a 1-interval. It is possible that zero, one or more such rollbacks may occur. In the following, we differentiate between the first such rollback and the subsequent rollbacks to the start of the segment; the reason will be explained below using an example. Let:

| | | |
|---|---|---|
| $S_c$ | $=$ | total time required to execute the above segment containing $(c+1)$ 1-intervals. $S_c$ is a random variable. |
| $P^@$ | $=$ | probability that a rollback will occur to the beginning of the segment, given that no previous rollback to the beginning of the segment has occurred. |
| $P$ | $=$ | probability that a rollback will occur to the beginning of the segment, given that at least one rollback to the beginning of the segment has already occurred. |
| $\rho$ | $=$ | number of times a rollback occurs to the beginning of the segment, *after* the first rollback to the beginning of the segment. $\rho$ is a random variable. |
| $F^@$ | $=$ | time lost due to a rollback to the beginning of the segment, given that this is the first rollback to the beginning of the segment. $F^@$ is a random variable. |
| $F$ | $=$ | time lost due to a rollback to the beginning of the segment, given that this is not the first rollback to the beginning of the segment. $F$ is a random variable. |
| $I^@$ | $=$ | time spent in executing a single 1-interval that terminates with a 1-checkpoint, given that at most a single failure occurs while executing the interval, and that a failure did not occur immediately before this interval started execution. $I^@$ is a random variable. |
| $I$ | $=$ | time spent in executing a single 1-interval that terminates with a 1-checkpoint, given that at most a single failure occurs while executing the interval, and that a failure occurred immediately before this interval started execution. $I$ is a random variable. |
| $I^{*@}$ | $=$ | time spent in executing a single 1-interval that terminates with an $N$-checkpoint, given that at most a single failure occurs while executing the interval, and that a failure did not occur immediately before this interval started execution. $I^{*@}$ is a random variable. |
| $E(x)$ | $=$ | expected value of random variable $x$. |

The appendix relates the above notation with a Markov chain representation of a segment's execution.

For accurate analysis, it is necessary to distinguish between the first rollback to the beginning of a segment and the subsequent rollbacks. Figure 4 illustrates this. As shown in the figure, length of the first 1-interval of the segment, before failures occur, is $T + C_1$. However, when a rollback to the start of the segment is needed, due to the additional $R$ time units required to rollback, the length of the first 1-interval in the segment is increased to $T+C_1+R$. (That is, we model the rollback overhead $R$ as a part of the 1-interval executed immediately following the failure.) After each subsequent rollback, the length of the first 1-interval is always $T + C_1 + R$.

The execution of a *segment* consists of two parts:

- Certain number of executions (may be zero or more) during which multiple failures occur that cause a rollback to the start of the segment: On the average, this requires $P^@ E(F^@) + P^@ E(\rho) E(F)$ units of time.
  **Justification:** $P^@$ is the probability that a rollback to the start of the segment will occur and $E(F^@)$ is the average cost of a *first* rollback to the start of the segment. Therefore, the first rollback contributes $P^@ E(F^@)$ to the average task completion time.
  $E(\rho)$ is the expected number of rollbacks to the start of the segment *after* the first such rollback. Therefore, the rollbacks to the start of the segment (excluding the first rollback) contribute $P^@ E(\rho) E(F)$ to the expected task completion time.

- An execution during which rollback to the start of the segment does not occur: On the average, this requires $(1 - P^@)E(I^@) + P^@ E(I) + (c-1)E(I^@) + E(I^{*@})$ units of time.

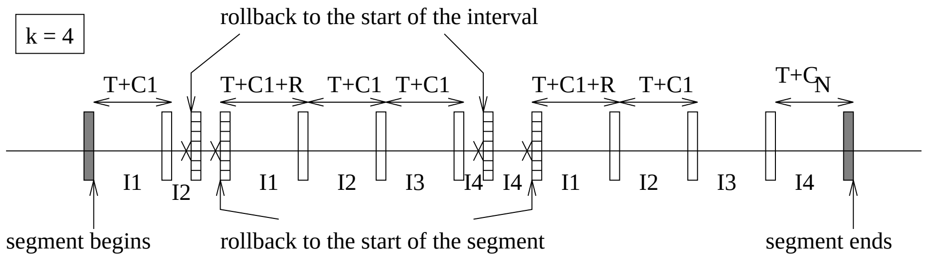


Figure 4: Rollback increases length of the first 1-interval

**Justification:** The expected time required to execute the first 1-interval after a rollback to the start of the segment is $E(I)$ and before such a rollback is $E(I^{@})$. Therefore, the expected time required to complete the first 1-interval of the segment is $(1 - P^{@})E(I^{@}) + P^{@}E(I)$. The expected time required to complete the last 1-interval of the segment is $E(I^{*@})$ and the expected time required to complete the middle $(c-1)$ 1-intervals is $(c-1)E(I^{@})$.

Therefore,

$$\begin{aligned} E(S_c) &= P^{@}E(F^{@}) + P^{@}E(\rho)E(F) + (1-P^{@})E(I^{@}) \\ &\quad + P^{@}E(I) + (c-1)E(I^{@}) + E(I^{*@}) \qquad (1) \end{aligned}$$

We first evaluate each quantity on the right hand side of the above equation. The reader may skip sections 3.3, 3.4 and 3.5 without loss of continuity.

## 3.3 Evaluation of $P^{@}$, $P$ and $E(\rho)$

We now define three probabilities (their definitions are similar). The appendix relates them with the Markov chain presented in the appendix.

$p^{@}$ = probability that a rollback to the start of the segment occurs during a given 1-interval that terminates with a 1-checkpoint, given that a failure did not occur immediately before this interval started.

$p$ = probability that a rollback to the start of the segment occurs during a given 1-interval that terminates with a 1-checkpoint, given that a failure occurred immediately before this interval started.

$p^{*@}$ = probability that a rollback to the start of the segment occurs during a given 1-interval that terminates with an $N$-checkpoint, given that a failure did not occur immediately before this interval started.

Then,

$$P^{@} = 1 - (1-p^{@})^c(1-p^{*@}).$$

A rollback will occur during a 1-interval if a processor fails before completion of the interval, and a processor also fails while re-executing the interval. Therefore,

$$p^{@} = (1 - e^{-N\lambda(T+C_1)})(1 - e^{-N\lambda(T+C_1+R)})$$

Similarly,

$$\begin{aligned} p^{*@} &= (1 - e^{-N\lambda(T+C_N)})(1 - e^{-N\lambda(T+C_N+R)}) \\ p &= (1 - e^{-N\lambda(T+C_1+R)})(1 - e^{-N\lambda(T+C_1+R)}) \end{aligned}$$

Knowing $p^{@}$ and $p^{*@}$, $P^{@}$ can be evaluated.

When it is known that at least one rollback occurred to the beginning of the segment, the length of the first 1-interval in the segment becomes $R + T + C_1$. The length of other 1-intervals is unchanged. Therefore,

$$P = 1 - (1-p)(1-p^{@})^{c-1}(1-p^{*@}), \qquad c > 0$$

It follows that $E(\rho) = P/(1-P)$.

## 3.4 Evaluation of $E(F)$

To be able to evaluate $E(F)$, we first need to evaluate $E(I^{@})$ and $E(I)$.

The definition of $I^{@}$ implies that a failure may occur while the 1-interval is executed, but no failure occurs when (and if) the 1-interval is re-executed. A rollback to the start of the 1-interval is required if a failure occurs any time during the $T$ units of execution or while taking the 1-checkpoint at the end of the 1-interval. Thus, a failure during $T + C_1$ time units can cause a rollback to the start of the 1-interval. When a rollback occurs, $R$ time units are spent in performing the rollback (i.e., initiating the re-execution). Therefore,

$$\begin{aligned} E(I^{@}) &= T + C_1 + \\ &\left[\frac{(1-e^{-N\lambda(T+C_1)})e^{-N\lambda(T+C_1+R)}}{e^{-N\lambda(T+C_1)} + (1-e^{-N\lambda(T+C_1)})e^{-N\lambda(T+C_1+R)}} \times \int_0^{T+C_1} (t+R)\frac{N\lambda\, e^{-N\lambda t}}{1-e^{-N\lambda(T+C_1)}}dt\right] \\ &= T + C_1 + \\ &\left[\frac{(1-e^{-N\lambda(T+C_1)})e^{-N\lambda(T+C_1+R)}}{e^{-N\lambda(T+C_1)} + (1-e^{-N\lambda(T+C_1)})e^{-N\lambda(T+C_1+R)}} \times \left(R + (N\lambda)^{-1} - \frac{(T+C_1)e^{-N\lambda(T+C_1)}}{1-e^{-N\lambda(T+C_1)}}\right)\right] \end{aligned}$$

If a failure occurs immediately before the start of a 1-interval that terminates with a 1-checkpoint, then length of that interval is $T + C_1 + R$. Therefore,

$$\begin{aligned} E(I) &= T + C_1 + R + \\ &\left[\frac{(1-e^{-N\lambda(T+C_1+R)})e^{-N\lambda(T+C_1+R)}}{e^{-N\lambda(T+C_1+R)} + (1-e^{-N\lambda(T+C_1+R)})e^{-N\lambda(T+C_1+R)}} \times \int_0^{T+C_1+R} (t)\,\frac{N\lambda\, e^{-N\lambda t}}{1-e^{-N\lambda(T+C_1+R)}}dt\right] \\ &= T + C_1 + R + \\ &\left[\frac{(1-e^{-N\lambda(T+C_1+R)})e^{-N\lambda(T+C_1+R)}}{e^{-N\lambda(T+C_1+R)} + (1-e^{-N\lambda(T+C_1+R)})e^{-N\lambda(T+C_1+R)}}\right. \end{aligned}$$

$$\times\left((N\lambda)^{-1} - \frac{(T+C_1+R)e^{-N\lambda(T+C_1+R)}}{1-e^{-N\lambda(T+C_1+R)}}\right)\Bigg]$$

Note that the integral term above contains $(t)$ unlike the integral term for $E(I^@)$ which contains $(t+R)$. This is because, for $E(I)$, $R$ is already included in the term outside the integral. $E(I^{*@})$ is obtained by replacing $C_1$ by $C_N$ in the equation for $E(I^@)$.

Recall that $F$ is defined as the time lost due to a rollback to the beginning of the segment, given that this is not the first rollback to the beginning of the segment. Evaluation of $E(F)$ is conditional on the fact that such a rollback indeed occurred. The rollback can occur during any one of the 1-intervals. Therefore,

$$E(F) \;=\; \sum_{i=1}^{c+1} Q_i\, E(F_i) \qquad \text{where,} \tag{2}$$

$Q_i$ is the probability that a rollback to start of the segment occurred during interval $i$ *given* that such a rollback occurred during the segment. $F_i$ is the execution time lost because of such a rollback during interval $i$. For $c > 0$,

$$Q_i = \begin{cases} p/P, & i=1 \\ (1-p)(1-p^@)^{i-2}p^@/P, & 1 < i \le c \\ (1-p)(1-p^@)^{c-1}p^{*@}/P, & i = c+1 \end{cases}$$

($p$, $p^@$, $p^{*@}$ and $P$ were obtained previously. It is easy to verify that $\sum_{i=1}^{c+1} Q_i = 1$.)

Given that a rollback to the start of the interval occurred during interval $i$, for $1 < i \le c$ and $c > 0$,

$$\begin{aligned} E(F_i) \;&=\; E(I) + (i-2)E(I^@) \\ &\quad + \int_0^{T+C_1} (t)\,\frac{N\lambda\, e^{-N\lambda t}}{1-e^{-N\lambda(T+C_1)}}dt \\ &\quad + \int_0^{T+C_1+R} (t)\,\frac{N\lambda\, e^{-N\lambda t}}{1-e^{-N\lambda(T+C_1+R)}}dt \\ &= E(I) + (i-2)E(I^@) + 2(N\lambda)^{-1} \qquad (3) \\ &- \frac{(T+C_1)e^{-N\lambda(T+C_1)}}{1-e^{-N\lambda(T+C_1)}} - \frac{(T+C_1+R)e^{-N\lambda(T+C_1+R)}}{1-e^{-N\lambda(T+C_1+R)}} \end{aligned}$$

$E(F_1)$ is obtained similar to $E(F_i)$.

$$\begin{aligned} E(F_1) \;&=\; 2\int_0^{T+C_1+R} (t)\,\frac{N\lambda\, e^{-N\lambda t}}{1-e^{-N\lambda(T+C_1+R)}}dt \\ &= 2(N\lambda)^{-1} - 2\frac{(T+C_1+R)e^{-N\lambda(T+C_1+R)}}{1-e^{-N\lambda(T+C_1+R)}} \end{aligned}$$

When $c > 0$, $E(F_{c+1})$ can be obtained by replacing $C_1$ by $C_N$ and $i$ by $c+1$ in Equation 3. $E(F)$ can now be evaluated using Equation 2 and the expressions for $E(F_i)$ and $Q_i$.

### 3.5 Evaluation of $E(F^@)$

Recall that $F^@$ is defined as the time lost due to a rollback to the beginning of the segment, given that this is the first rollback to the beginning of the segment. Evaluation of $E(F^@)$ is very similar to the evaluation of $E(F)$.

$$E(F^@) \;=\; \sum_{i=1}^{c+1} Q_i^@\, E(F_i^@) \qquad \text{where,} \tag{4}$$

$Q_i^@$ is the probability that a rollback to start of the segment occurred during interval $i$ *given* that such a rollback occurred during the segment and that this is the first such rollback in this segment. $F_i^@$ is the execution time lost because of such a rollback during interval $i$. For $c > 0$,

$$Q_i^@ = \begin{cases} (1-p^@)^{i-1}p^@/P^@, & 1 \le i \le c \\ (1-p^@)^c p^{*@}/P^@, & i = c+1 \end{cases}$$

Note that $\sum_{i=1}^{c+1} Q_i^@ = 1$.

Given that a rollback to the start of the interval occurred during interval $i$, for $1 \le i \le c$ and $c > 0$,

$$\begin{aligned} E(F_i^@) = (i-1)E(I^@) &+ \int_0^{T+C_1} t\frac{N\lambda\, e^{-N\lambda t}}{1-e^{-N\lambda(T+C_1)}}dt \\ &+ \int_0^{T+C_1+R} t\,\frac{N\lambda\, e^{-N\lambda t}}{1-e^{-N\lambda(T+C_1+R)}}dt \end{aligned} \tag{5}$$

$E(F_{c+1}^@)$ can be obtained by replacing $C_1$ by $C_N$ and $i$ by $c+1$ in Equation 5. $E(F^@)$ can now be evaluated using Equation 4 and the expressions for $E(F_i^@)$ and $Q_i^@$.

### 3.6 Evaluation of expected task completion time

Using the expressions derived above and Equation 1, the value of $E(S_c)$, $c > 0$, can now be evaluated. $E(S_c)$ for $c = 0$ can also be obtained similarly. Recall that length of the task ($\Upsilon$) is an integral multiple of $T$. Specifically, $\Upsilon = \mu T$. The task consists of $\lceil \mu/k \rceil$ segments, of which $\lceil \mu/k \rceil - 1$ segments contain $k$ 1-intervals each and one segment contains $k^{\#} = \mu - k\,(\lceil \mu/k \rceil - 1)$ 1-intervals. Therefore, the expected task completion time $E(\Gamma)$ is obtained as

$$E(\Gamma) \;=\; (\lceil \mu/k \rceil - 1)\, E(S_{k-1}) + E(S_{k^{\#}-1}) \tag{6}$$

As we know how to evaluate $E(S_c)$ for arbitrary $c$, the expected task completion time can now be evaluated.

### 3.7 Average Performance Overhead

The average performance overhead can be obtained as,

$$\begin{aligned} \text{average overhead} \;&=\; \frac{E(\Gamma)}{\Upsilon} - 1 \\ &= \frac{(\lceil \mu/k \rceil - 1)\, E(S_{k-1}) + E(S_{k^{\#}-1})}{\mu T} - 1 \end{aligned} \tag{7}$$

Average *percentage* overhead is obtained by multiplying the average overhead by 100. For very large tasks ($\mu \to \infty$), the average overhead approaches $\frac{E(S_{k-1})}{kT} - 1$.

## 4 Numerical Results

In this section, we present numerical results to determine optimal values of $k$ and $\mu$ that minimize the average overhead, for a given task size and a given $\lambda$. Significant effort has been devoted in the past for analytically determining optimal checkpoint intervals for checkpointing and rollback recovery schemes (e.g., [4, 7, 19]). Due to the complexity of the expressions for the two-level recovery scheme under consideration, an analytical approach for determining optimal $k$ and $\mu$ is not very attractive. Instead, we choose to determine the optimal values numerically.

A number of parameters affect the performance overhead, including $C_1$, $C_N$, $\lambda$, $N$ and task length $\Upsilon$. In this paper, we are primarily interested in the effect of relative values of $C_1$ and $C_N$ on the optimal operating point. (For a given task, an operating point is characterized by the chosen values of $k$ and $\mu$.)

We evaluate the average overhead for a hypothetical task characterized by following parameters: $\lambda = 0.00001$ per time unit, $\Upsilon = 200$ time unit, $N = 500$, $C_N = 1.0$ time unit, $R = 1.0$ time unit. Different values of $C_1$ are used in the following for different graphs.

Due to the limitations of our graph-plotting software, $\mu$ is denoted as `MU` in the graphs.

## "Non"-Convex Curves for Two-Level Recovery

The first interesting feature of the two-level scheme is that the performance overhead curves do not always have a unique minimum. Figure 5 plots the average percentage overhead versus $\mu$ for $C_1 = 0.2$ time unit and $k = 3$ and 10. Observe that the curves for $k = 3$ and 10 have multiple minima. These curves are not convex, unlike the traditional checkpointing and rollback schemes (e.g., [4]).

The curve for $k = 1$ and $C_1 = 0.2$ is shown in Figure 6. When $k = 1$, the two-level recovery scheme reduces to the traditional checkpointing and rollback scheme that takes only $N$-checkpoints. Therefore, as shown previously in [4], the curve for $k = 1$ is convex and has exactly one minimum. Note that the curve for $k = 1$ is independent of $C_1$, as no 1-checkpoints are taken in this case.

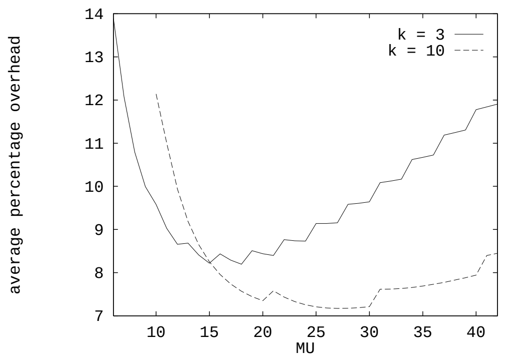


Figure 5: $C_1 = 0.2$ and $k = 3$, 10 – curves are not convex

## Optimization of the Two-Level Recovery Scheme

Figures 7 through 10 plot the percentage overhead versus $\mu$ for various values of $k$ and $C_1$. For four different values of $C_1$, we evaluated the average overhead for various values of $k$ and $\mu$, and determined the optimal values of $k$ and $\mu$ that minimize the average percentage overhead. The optimal values of $k$ and $\mu$ are presented in Table 1. Note that when $C_1 = 1.0$ (see Figure 10), we have $C_1 = C_N$, i.e., taking 1-checkpoints is as expensive as $N$-checkpoints. As $N$-checkpoints provide more protection against failures, it is obvious that, to minimize the average overhead, all the checkpoints must be $N$-checkpoints (i.e., $k = 1$). In practice, $C_1$ will often be smaller than $C_N$.

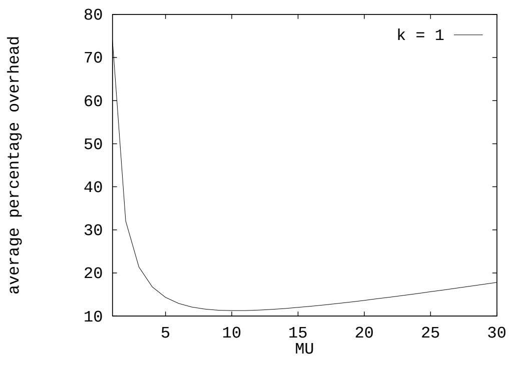


Figure 6: $k = 1$ – the curve is convex

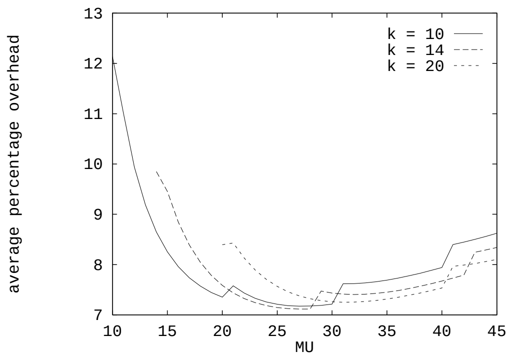


Figure 7: $C_1 = 0.2$

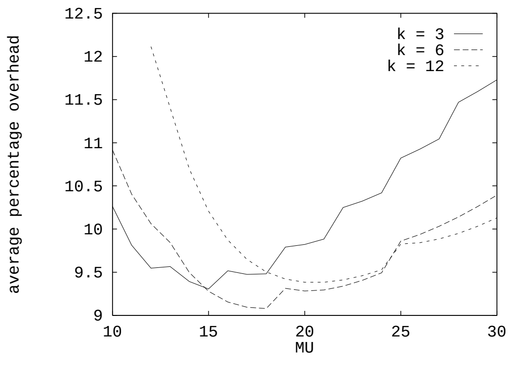


Figure 8: $C_1 = 0.4$

| $C_1$ | $k$ | $\mu$ | average % overhead |
|---|---|---|---|
| 0.2 | 14 | 27 | 7.1 |
| 0.4 | 6 | 18 | 9.1 |
| 0.6 | 3 | 14 | 10.3 |
| 1.0 | 1 | 10 | 11.2 |

Table 1: Minimum average percentage overhead

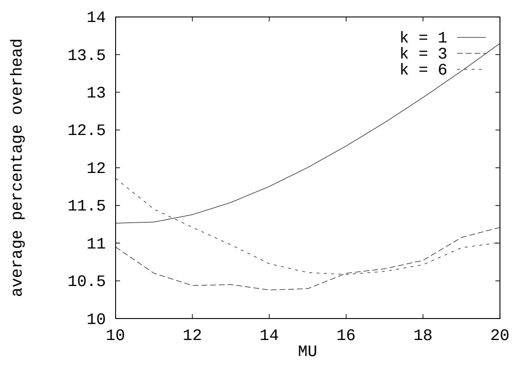


Figure 9: $C_1 = 0.6$

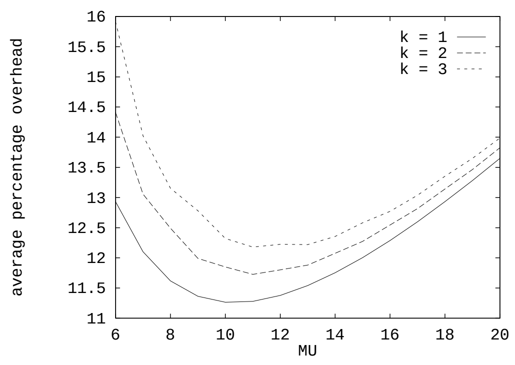


Figure 10: $C_1 = 1.0$

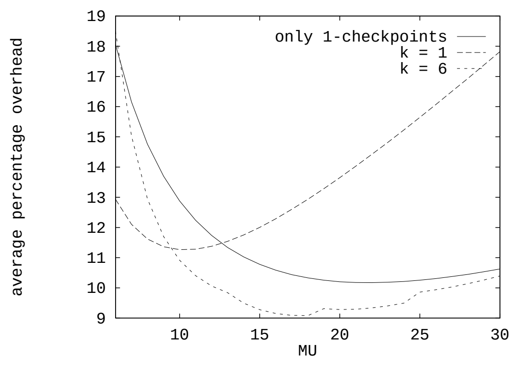


Figure 11: $C_1 = 0.4$

## Comparison With One-Level Recovery Schemes

Two one-level recovery schemes (corresponding to the two component schemes) are compared with the two-level recovery scheme.

The first one-level scheme is the traditional consistent checkpointing scheme that takes only $N$-checkpoint. When $k = 1$, the two-level recovery scheme is identical to this one-level scheme. In the graphs presented above, observe that, with the exception of the case when $C_1 = C_N = 1.0$, the performance overhead is minimized when $k > 1$. This implies that, the two-level recovery scheme can achieve a lower performance overhead as compared to the one-level recovery scheme except when $C_1 = C_N$. When $C_1 = C_N$, the 1-checkpoints are as expensive as $N$-checkpoints, and therefore, the two-level recovery scheme can only perform as well as the one-level scheme.

The second one-level scheme takes only 1-checkpoints, i.e., all $\mu$ checkpoints are 1-checkpoints. When multiple failures occur within a single 1-interval, this recovery scheme requires that the task be restarted from the beginning. (This scheme may also be viewed as a degenerate two-level scheme, as it tolerates single and multiple failures differently.) To compare the performance of the second one-level scheme with the two-level scheme, Figure 11 shows the curves for the two-level scheme with $k = 6$ and for the one-level scheme that takes only 1-checkpoints. (The figure also plots the curve for $k = 1$, i.e., for the first one-level scheme.) Observe that the two-level scheme can achieve a lower performance overhead as compared to the scheme that takes only 1-checkpoints.

In the above we have compared three schemes: the two-level schemes, and the two component schemes. It should be noted that, for each scheme, it is possible to pick a set of parameters for which the chosen scheme will perform better than the other two. However, our numerical search suggests that the one-level scheme that takes only $N$-checkpoints is *not* optimal for many sets of parameters (which we believe to be realistic parameters). This is interesting, as many current implementations of consistent checkpoints take only $N$-checkpoints, and therefore are not likely to be optimal for many applications.

## 5 Related Work

We define *two-level* recovery schemes as those that tolerate *more probable* failures with a low overhead, while the *less probable* failures may incur a higher overhead. This definition can also be extended to *multi-level* schemes.

It was recently brought to our attention [3] that Gelenbe [8] previously proposed a "multiple checkpointing" approach that is similar to the "multi-level" approach that we advocate in this paper. Gelenbe divides system failures into multiple ($n$) categories according to their severity. The system takes $n$ types of checkpoints, each type of checkpoint designed for one type of failure. Each type of failure is assumed to be governed by a Poisson process. Although Gelenbe considers transaction-oriented systems, the fundamental idea behind *multiple checkpoints* and *multi-level recovery* is the same – minimize overhead by designing different approaches for tolerating different types of failures. We characterize a failure "type" according to the probability of its occurrence, while Gelenbe characterizes a failure "type" according to how "difficult" it is to recover from the failure. (A failure of type 1 is less "difficult" than a failure of type 2 if a checkpoint for failure type 2 can be used to recover from a type 1 failure [8].) Gelenbe's analysis as such may not be applicable to multi-level schemes of our interest, for three reasons:

- Our multi-level approach is *not* confined to multiple types of checkpoints. The component schemes can, for example, use message logging. When message logging is used, the failure-free overhead may increase as the length of the interval between checkpoints increases. On the other hand, when only checkpoints are taken, the overhead of a single checkpoint is typically assumed to be independent of the length of checkpoint interval.

- Gelenbe assumes the failures of different types to be governed by Poisson process. This may not be true, in general, even if the failure of each processor is governed by a Poisson process. For instance, this assumption will **not** apply for the two-level scheme presented in this paper.

- Gelenbe considers transaction-oriented systems. The analysis for a distributed system executing a long-running parallel application may differ (depending on the multi-level scheme under consideration).

Ziv and Bruck [21] present a checkpointing scheme for *duplex* systems. Although it does not satisfy our definition of two-level schemes, their scheme also takes two types of checkpoints. They assume that the duplex system is formed by a pair of workstations connected by a local area network (LAN). It is assumed that the state of the two processors in a duplex system must be compared to detect failures. To compare the checkpoints, the checkpoints must be sent over the LAN. The overhead of checkpoint comparison, therefore, is high as compared to saving the checkpoints (the checkpoints are saved on the local disk of each workstation). Ziv and Bruck propose a scheme where checkpoint comparison is performed only at every $k$-th checkpoint. If a failure is detected, then the previous $k$ checkpoints are compared until an error-free checkpoint is found. The duplex system then rolls back to this checkpoint. By restricting checkpoint comparison (during failure-free operation) to every $k$-th checkpoint, [21] reduces overhead of the recovery scheme, as compared to a scheme that compares the states at *each* checkpoint. Our approach differs from [21] in that we attempt to minimize the average overhead by distinguishing between *more probable* and *less probable* failures. [21] improves the overhead (for duplex systems) by decoupling *checkpoint saving* and *checkpoint comparison*.

We previously proposed a *roll-forward* scheme [14], for duplex systems, that tolerates single processor failures with a low overhead, and multiple failures with a high overhead. This is achieved by taking different actions during recovery, depending on the number of failures – the actions taken during failure-free operation are independent of the number of expected failures. Although this scheme satisfies our definition of *two-level* recovery, our present research is concerned with recovery schemes that take *explicit* actions during *failure-free* operation that are designed to minimize the overhead for the *more probable* failures.

## 6 Conclusions

The objective of this paper was to illustrate the two-level recovery approach using a simple example, and to show that two-level recovery scheme can achieve better performance than traditional recovery schemes. The paper presented a two-level recovery scheme that tolerates single failures with a low overhead and multiple failures with a higher overhead. From the performance analysis of this two-level recovery scheme, the following conclusions can be drawn:

- To minimize the overhead, it is often necessary to take both 1-checkpoints and $N$-checkpoints. This implies that the two-level recovery scheme can achieve lower performance overhead as compared to the one-level recovery schemes.

- The optimal values of $\mu$ and $k$ at which the overhead is minimized are sensitive to the changes in $C_1$. Changes in $C_1$ can affect the performance overhead significantly. Therefore, it is desirable to keep $C_1$ as small as possible.

The above conclusions motivate the following research:

- Techniques to reduce the overhead of single failure tolerance need to be investigated. In general, it is necessary to design efficient recovery schemes to tolerate the *more probable* failure scenarios. Past research on such recovery schemes is very limited [1, 2, 9, 10]. Another important question is how to characterize the *more probable* failures such that the performance is optimized (or, for multi-level schemes, how to characterize the different *levels*). These characterizations should take the hardware organization into account. In this paper, somewhat arbitrarily, we characterize single failures as the *more probable* failures.

- Design of other two-level recovery schemes for message passing and shared memory systems. Although the above conclusions imply that two-level recovery can achieve better performance as compared to traditional one-level recovery schemes, research on such schemes is lacking. This paper provides a strong motivation for further research on two-level recovery.

- Analytical optimization of two-level schemes. In this paper, the optimal was determined by numerical search.

- Implementation and experimental evaluation of two-level recovery schemes. Analysis of distributed failure recovery schemes is usually approximate, because parameters such as checkpoint overhead are time-dependent and difficult to characterize accurately. Therefore, an experimental evaluation is desirable.

**Acknowledgements**

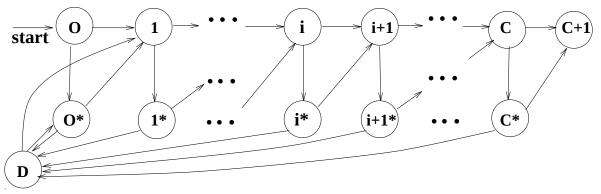
We thank the referees for pointing us to the work by Gelenbe [8], and for other helpful comments.

## Appendix

The Markov chain below models execution of a single segment. State $i$ ($i > 0$) in the Markov chain is entered when the $i$-th checkpoint in the segment is established. State $i^*$ ($i \geq 0$) is entered when a failure occurs while in state $i$ (i.e., when the first failure occurs while executing $i+1$-th interval). State $D$ is entered when a failure occurs while in state $i^*$ before the $i+1$-th 1-interval is completed (i.e., when double failures occur while executing an interval).

Transitions out of state $D$ are similar to those from state 0. (Transitions into states $D$ and $i^*$, $0 \leq i \leq c$ correspond to rollbacks.)

A *cost* is associated with each transition. The *cost* of a transition (X,Y) from state X to Y is the expected time spent in state X before making the transition to state Y. For example, for $0 \leq i < c$, the cost of the transition (i,i+1) is $T + C_1$, the cost of transition (i$^*$,i+1) is $R + T + C_1$, the cost of transition (i,i$^*$) is $(N\lambda)^{-1} - \frac{(T+C_N)e^{-N\lambda(T+C_N)}}{1-e^{-N\lambda(T+C_N)}}$. Similarly, the cost of transition $(c, c+1)$ is $T + C_N$.



*Transition probability* for transition (X,Y) is the probability that a transition to state Y will be made from state X. For instance, for $0 \leq i < c$, transition probability for transition (i,i+1) is $e^{-N\lambda(T+C_1)}$, for transition (i,i$^*$) is $1 - e^{-N\lambda(T+C_1)}$, and for transition (i$^*$,D) is $1 - e^{-N\lambda(R+T+C_1)}$.

The execution of the segment can be viewed as following a path from state 0 to state $c+1$. $E(S_c)$ is, then, the expected cost of a path from state 0 to state $c+1$. $E(S_c)$ can be evaluated using standard techniques for Markov chains [16, 17, 20].

Now we relate the notation in Section 3 to the above Markov chain. $P^@$ is the probability that a path from state 0 will reach state $D$. $P$ is the probability that a path originating at state $D$ will return to state $D$. $\rho$ is the number of entries into state $D$ excluding the first entry into state $D$. $F^@$ is the length of a path from state 0 to state $D$. $F$ is the length of a path from state $D$ back to state $D$. $I^@$ is the length of a path, from state $i$ ($0 \leq i < c$) to state $i+1$, that does not enter state $D$. $I$ is the length of a path, from state $D$ to state 1, that does not re-enter state $D$. $I^{*@}$ is the length of a path, from state $c$ to state $c+1$, that does not enter state $D$. $p^@$ is the probability that a transition will be made from state $i$ ($0 \leq i < c$) to state $i^*$, followed by a transition from state $i^*$ to state $D$. $p^{*@}$ is the probability that a transition will be made from state $c$ to state $c^*$, followed by a transition from state $c^*$ to state $D$. $p$ is the probability that a transition will be made from state $D$ to state $0^*$, followed by a transition from state $0^*$ to state $D$.

## References

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

[2] L. Alvisi and K. Marzullo, "Optimal message logging protocols," Tech. Rep. in preparation, Department of Computer Science, Cornell University, 1994.

[3] Anonymous referee's comments on this paper, January 1995.

[4] K. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig, "Analytic models for rollback and recovery strategies in data base systems," *IEEE Trans. Softw. Eng.*, vol. 1, pp. 100–110, March 1975.

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

[6] S. Garg and K. F. Wong, "Analysis of an improved distributed checkpointing algorithm," WUCS-93-37, Dept. of Comp. Sc., Washington Univ., June 1993.

[7] E. Gelenbe and D. Derochette, "Performance of rollback recovery systems under intermittent failures," *Comm. ACM*, vol. 21, pp. 493–499, June 1978.

[8] E. Gelenbe, "A model for roll-back recovery with multiple checkpoints," in *2nd Int. Conf. on Software Engineering*, pp. 251–255, October 1976.

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

[10] J. León, A. L. Fisher, and P. Steenkiste, "Fail-safe PVM: A portable package for distributed programming with transparent recovery," Tech. Rep. CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, Pittsburgh, February 1993.

[11] V. F. Nicola and J. M. van Spanje, "Comparative analysis of different models of checkpointing and recovery," *IEEE Trans. Softw. Eng.*, pp. 807–821, August 1990.

[12] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, 1994.

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

[14] D. K. Pradhan and N. H. Vaidya, "Roll-forward checkpointing scheme: A novel fault-tolerant architecture," *IEEE Trans. Computers*, pp. 1163–1174, Oct. 1994.

[15] A. N. Tantawi and M. Ruschitzka, "Performance analysis of checkpointing strategies," *ACM Trans. Comp. Syst.*, vol. 2, pp. 123–144, May 1984.

[16] K. S. Trivedi, *Probability and Statistics with Reliability, Queueing and Computer Science Applications.* Prentice-Hall, 1988.

[17] N. H. Vaidya, "Another *two-level* failure recovery scheme: Performance impact of checkpoint placement and checkpoint latency," Tech. Rep. 94-068, Computer Science, Texas A&M University, Dec. 1994.

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

[19] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Comm. ACM*, vol. 17, pp. 530–531, September 1974.

[20] A. Ziv and J. Bruck, "Analysis of checkpointing schemes for multiprocessor systems," Tech. Rep. RJ 9593, IBM Almaden Research Center, Nov. 1993.

[21] A. Ziv and J. Bruck, "Efficient checkpointing over local area network," in *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, College Station, June 1994.