Proactive management of software aging

by V. Castelli
R. E. Harper
P. Heidelberger
S. W. Hunter
K. S. Trivedi
K. Vaidyanathan
W. P. Zeggert

Software failures are now known to be a dominant source of system outages. Several studies and much anecdotal evidence point to "software aging" as a common phenomenon, in which the state of a software system degrades with time. Exhaustion of system resources, data corruption, and numerical error accumulation are the primary symptoms of this degradation, which may eventually lead to performance degradation of the software, crash/hang failure, or other undesirable effects. "Software rejuvenation" is a proactive technique intended to reduce the probability of future unplanned outages due to aging. The basic idea is to pause or halt the running software, refresh its internal state, and resume or restart it. Software rejuvenation can be performed by relying on a variety of indicators of aging, or on the time elapsed since the last rejuvenation. In response to the strong desire of customers to be provided with advance notice of unplanned outages, our group has developed techniques that detect the occurrence of software aging due to resource exhaustion, estimate the time remaining until the exhaustion reaches a critical level, and automatically perform proactive software rejuvenation of an application, process group, or entire operating system, depending on the pervasiveness of the resource exhaustion and our ability to pinpoint the source. This technology has been incorporated into the IBM Director for xSeries servers. To quantitatively evaluate the impact of different rejuvenation policies on the availability of cluster systems, we have developed analytical models based on stochastic reward nets (SRNs). For time-based rejuvenation policies, we determined the optimal rejuvenation interval based on system availability and cost. We also analyzed a rejuvenation policy based on prediction, and showed that it can further increase system availability and reduce downtime cost. These models are very general and can capture a multitude of cluster system characteristics, failure behavior, and performability measures, which we are just beginning to explore.

1. Introduction

Software aging

Unplanned computer system outages are more likely to be the result of software failures than of hardware failures [1, 2]. Moreover, software often exhibits an increasing failure rate over time, typically because of increasing and unbounded resource consumption, data corruption, and numerical error accumulation. This constitutes a

0018-8646/01/$5.00 © 2001 IBM

phenomenon called *software aging* [3], and may be caused by errors in the application, middleware, or operating system. Under aging conditions, the state of the software degrades gradually with time, inevitably resulting in undesirable consequences. Some typical causes of this degradation are memory bloating and leaking, unterminated threads, unreleased file-locks, data corruption, storage-space fragmentation, and accumulation of round-off errors. This phenomenon has been reported by Huang et al. [3] in telecommunications billing applications, where over time the application experiences a crash or a hang failure. Avritzer and Weyuker discuss aging in telecommunication switching software, in which the effect manifests itself as gradual performance degradation [4]. Software aging has been observed not only in specialized software, but also in widely used software, where rebooting to clear a problem is a common practice.

Aging occurs because software is extremely complex and never wholly free of errors. It is almost impossible to fully test and verify that a piece of software is bug-free. This situation is further exacerbated by the fact that software development tends to be extremely time-to-market-driven, which results in applications which meet the short-term market needs, yet do not account very well for long-term ramifications such as reliability. Hence, residual faults have to be tolerated in the operational phase. These residual faults can take various forms, but the ones that we are concerned with cause long-term depletion of system resources such as memory, threads, and kernel tables. The essentially economic problem of developing and producing bug-free code is not the problem at hand; instead we address one of the problems that arises from the prevailing approach to developing software, and one approach to attacking that problem is software rejuvenation.

Software rejuvenation

To counteract software aging, a proactive technique called *software rejuvenation* has been devised [3]. It involves stopping the running software occasionally, "cleaning" its internal state (e.g., garbage collection, flushing operating system kernel tables, and reinitializing internal data structures) and restarting it. An extreme but well-known example of rejuvenation is a system reboot. There are numerous examples in real-life systems where software rejuvenation is being used. For example, it has been implemented in the real-time system collecting billing data for most telephone exchanges in the United States [5]. *Software capacity restoration*, a technique similar to rejuvenation, has been used by Avritzer and Weyuker in a large telecommunications-switching software application [4]. In this case, the switching computer is rebooted occasionally, which restores its service rate to the peak value. Grey [6] proposed performing operations solely for fault management in Strategic Defense Initiative (SDI) software which are invoked whether or not the fault exists, and called it operational redundancy. Tai et al. [7] have proposed and analyzed the use of onboard preventive maintenance for maximizing the probability of successful mission completion for spacecraft with very long mission times. The necessity of performing preventive maintenance in a safety-critical environment is evident from the example of aging in Patriot missile software [8]. The failure, which resulted in loss of human lives, might have been prevented had the operators heeded the advice that the system had to be restarted after every eight hours of running time. The Apache Web Server[1] (from The Apache Software Foundation) provides a means to prevent itself from becoming too much of a resource burden on a system. Apache has a controlling process and a handler process. The controlling process watches the handler process to ensure that it is running up to standard. The handler process, on the other hand, handles requests from the clients. When the handler process is deemed to be in a bad state, the controlling process stops it and starts another process.

Most current fault-tolerance techniques are *reactive* in nature. *Proactive fault management*, on the other hand, takes suitable corrective action to prevent a failure *before* the system experiences a fault. Although this technique has long been used on an *ad hoc* basis in physical systems, it has only recently gained recognition and importance for computer systems. Software rejuvenation is a specific form of proactive fault management which can be performed at suitable times, such as when there is no load on the system, and thus typically results in less downtime and cost than the reactive approach. Since proactive fault management incurs some overhead, an important research issue is to determine the optimal times to invoke it in operational software systems. Proactive fault management can be greatly enhanced by the ability to predict the fault far enough in advance that one can take action to avoid or mitigate its effects. Resource exhaustion by its very nature offers clues that failure is imminent, in the form of parameters that can be monitored, extrapolated, and compared to thresholds via suitable algorithms.

Software failure prediction and rejuvenation in a cluster environment

A cluster is a collection of independent, self-contained computer systems working together to provide a more reliable and powerful system than a single node by itself [9]. Clustering has proven to be an effective method of scaling to larger systems for added performance, more users, or other attributes [10], as well as providing higher

[1] The Apache Group, Apache http Server Project, *http://www.apache.org.*

levels of availability and lower management costs. One of the benefits of clustering is its natural redundancy of hardware and software components. A number of single points of failure can be removed by cluster systems. As part of a clustered system, a "failover" process is usually employed, which transfers workload to another portion of the clustered system when a hardware or software failure occurs. An objective of the failover process is for it to occur gracefully, without an end user knowing that a failure has occurred. In practice, the successful achievement of this objective is highly application-dependent. Also, in order to ensure the ability to perform a failover, sufficient spare resources must be available to accommodate the migrated workload. Another advantage of a clustered system is its ability to improve system maintenance. For example, if a specific resource of a cluster is in need of repair and a spare resource is available, it is possible, in a planned manner, to move the load from the resource being repaired, perform a shutdown, and remove and replace the resource, if necessary.

Software rejuvenation technology is a natural fit with clustered systems. Within a clustered environment, rejuvenation can be performed by invoking the cluster's failover mechanisms, either on a periodic basis based on prior experience of the time to resource exhaustion, or extemporaneously, upon prediction of an impending resource exhaustion. Our analyses (discussed in the following sections) show that combining software rejuvenation with clustering significantly increases system availability. Using the node failover mechanisms in a high-availability cluster, one can maintain operation (though possibly at a degraded level) while rejuvenating one node at a time, assuming that a node rejuvenation takes less time to perform and is far less disruptive than recovering from an unplanned node failure. Because the user's application has presumably been written to survive node failovers anyway, this environment has the added advantage of allowing rejuvenation to be transparent to the application. Simple time-based rejuvenation policies, in which the nodes are rejuvenated at regular intervals, can be implemented easily, using the existing cluster-management infrastructure. System availability can be further enhanced by taking a proactive approach to detect and predict an impending outage of a specific server in order to initiate planned failover in a more orderly fashion. This approach not only improves the end user's perception of service provided by the system, but also gives the system administrator additional time to work around any system capacity issues that may arise.

Because of these attractive possibilities, we have implemented the first commercial version of the xSeries* Software Rejuvenation Agent (SRA) within a high-availability clustered environment, which is being managed by the IBM Director system management tool. We are also in the process of analyzing and implementing the use of software rejuvenation within other applications such as large web-server pools, but results are not yet available.

Main contribution and outline

The main contribution of this paper is the development of a methodology for proactive management of software systems which are prone to aging, and specifically to resource exhaustion. The application of software rejuvenation for cluster systems is by itself a novel contribution. We have designed and developed a rejuvenation agent for IBM Director which manages a highly available clustered environment. Several algorithms for prediction of resource exhaustion, which is an important component of this methodology, are discussed. Rejuvenation incurs some overhead in terms of both downtime and cost, and if done more often than necessary will result in higher downtime and/or cost. Hence, stochastic models of the cluster system are developed and analyzed for some rejuvenation policies. For the time-based policies, we demonstrate the effect of the rejuvenation interval (defined as the time between successive rejuvenations) on the steady-state expected downtime and cost. We then obtain the optimal value of this interval, which minimizes the downtime and cost for an assumed set of parameter values. These models are very general and can capture a multitude of cluster system characteristics, failure behavior, and performability measures.

The remainder of this paper is organized as follows. Section 2 discusses previous related work and outlines the novel aspects of our work. The xSeries Software Rejuvenation Agent, the architecture of IBM Director, and how rejuvenation is performed in this framework, are described in Section 3. The various statistical algorithms used for prediction are also discussed in this section. Section 4 deals with stochastic models for cluster systems and analysis of some rejuvenation policies. The models capture several cluster system characteristics, and rejuvenation is performed in a cluster environment. Since rejuvenation incurs an overhead, these models are helpful in comparing several rejuvenation policies and optimizing them for maximum benefit in terms of cost and availability. Section 5 discusses experimental studies of software aging for some common applications. Empirical data on their aging characteristics are obtained and studied. These studies are essential in understanding important parameters to monitor and in developing application-specific rejuvenation strategies. Conclusions and possible extensions to our work are discussed in Section 6.

2. Related work

Garg et al. [11] present a general methodology for detecting and estimating trends and times to exhaustion of operating system resources due to software aging. In their work, data on system activity and resource usage was collected at regular intervals using a simple network management protocol (SNMP)-based tool. Whereas in [11] only time-based trend detection and estimation of resource exhaustion are considered, [12] takes the system workload into account for building a model. Other work in measurement-based dependability evaluation is based on measurements made either at failure times [13–15] or at error observation times [16–18]. In [11] and [12], the system parameters are monitored continuously, since the evaluation is interested in trend estimation and not in interfailure times or identifying error patterns. Some of the above papers deal with hardware failures, while [11] and [12] are concerned solely with software failures—in particular, failures due to resource exhaustion.

Iyer and Rossetti [14] study the effect of system workload on failures through a measurement-based analysis and report significant correlations between permanent failures and increased system activity. A methodology for recognizing the symptoms of persistent problems is proposed in [16], which identifies and statistically validates recurring patterns among error records produced in the system. In [19], system parameters are constantly monitored to detect anomalies automatically. This procedure is based on the premise that an anomaly or a deviation from "normal" behavior is usually a symptom of error. This study focuses on network failures, and smoothing techniques are applied to build a normal behavior profile.

Several analytical studies have assessed the effectiveness of software rejuvenation. In [3, 7, 20–22] only failures causing unavailability of the software are considered, while in [23] only a gradually decreasing service rate of a software application which serves transactions is assumed. In [24], however, both these effects of aging are considered together in a single model. Models proposed in [3, 20, 21] are restricted to hypoexponentially distributed time to failure. Our analysis is very similar in that it uses hypoexponentially distributed time to failure for the software; however, in [20], Markov regenerative theory is used to solve the model, whereas we use an Erlang approximation. The models proposed in [7, 22, 23] can accommodate general distributions, but only for the specific aging effect they capture. Generally distributed time to failure, and the service rate being an arbitrary function of time, are allowed in [24]. It has been noted [2] that transient failures are partly caused by overload conditions, but only the model presented in [24] captures the effect of load on aging. In [25], an availability model of a two-node cluster is described. Different failure scenarios are modeled, and the availability is analyzed. While that paper concentrates on hardware failures, and software rejuvenation is not considered, this paper considers rejuvenation with software failures only. Existing models also differ in the measures being evaluated and the assumptions underlying the analysis. For example, in [7, 22], software with a finite mission time is considered, whereas we consider continuously running systems. In [3, 20, 21, 24], measures of interest in transaction-based software intended to run forever are evaluated; we analyze availability and cost. All previous models except [7] and [22] are just special cases of the model presented in [24].

In [25], an availability model of a two-node management scheduling and control system (MSCS) is described. Different failure scenarios are modeled, and availability is analyzed. A cluster system modeled with working, failed, and intermediate states is described in [26]. Reliability analysis is carried out for a telecommunication system application for a range of failure-rate and fault-coverage values. In [27], hardware, operating system, and application software reliability techniques are discussed for the modeled cluster systems. Reliability levels in terms of fault detection, fault recovery, volatile data consistency, and persistent data consistency are described. While the above papers model hardware and software failures, and software rejuvenation is not considered, our modeling and analysis consider rejuvenation with software failures only.

3. The xSeries Software Rejuvenation Agent

The xSeries Software Rejuvenation Agent (SRA) was designed to monitor consumable resources, estimate the time to exhaustion of those resources, and generate alerts to the management infrastructure when the time to exhaustion is less than a user-defined notification horizon. The management infrastructure provides a graphical user interface for the user to configure the SRA, and accepts and acts upon the alerts as described below.

The SRA was designed according to a number of ground rules, with the prime objective of maximizing flexibility, portability, and customer acceptance. For maximum generality and user acceptance, no modification to the application is allowed, to support either failure prediction or rejuvenation. Similarly, no access to the kernel is allowed, in order to facilitate error containment, cross-OS portability, and customer acceptance. The agent must use published and architected interfaces for data acquisition, alerting, and rejuvenation in order to minimize sensitivity to gratuitous interface churn and provide a product that is relatively stable across multiple generations of operating systems and applications. The agent must be relatively portable across operating systems to allow us to economically attack the different markets of

commercial interest to IBM. A simple user interface is required which contains a minimum number of tunable parameters and is easy to set up and understand. Finally, because in many cases we do not know in advance which resources will be exhausted in the myriad environments in which we will be using SRA, the agent must be able to adapt to monitor new exhaustible parameters and execute new algorithms to predict exhaustion of those resources. These considerations caused us to partition the design into an OS-dependent data acquisition subsystem, a portable analytical subsystem with highly configurable input parameters, an architected management interface, and a management infrastructure that spans multiple IBM-supported operating systems. The SRA is incorporated as a component of IBM Director, which is discussed next.

Architecture and framework of IBM Director

Tivoli provides various systems management options for large-size as well as medium-size companies. Their technology is leveraged by xSeries for IBM Director and provides the xSeries brand with the framework of a high-quality systems management toolkit, while allowing us the flexibility to develop value-add functionality such as the SRA. IBM Director is a three-tier environment (**Figure 1**) that supports this flexibility on top of a highly scalable infrastructure. This cross-platform flexibility allows the xSeries brand to meet customer needs, whether they are using a Microsoft** operating system, Novell**, OS/2*, SCO**, or Linux.

The three tiers of IBM Director are the console, server, and agent. The console provides a Java**-based interface for accessing the functionality (via a set of tasks) of the IBM Director environment. The server controls access to the function, data, and agents for a given task, and the agent is the interface to a managed object, which in our case is a server or cluster of xSeries servers. Events provide a notification mechanism from an agent into the IBM Director environment. IBM Director is designed to operate in a client/server model and consists of the following components:

- *Management console*: The IBM Director management console is the graphical user interface (GUI) from which administrative tasks are performed. It is the primary interface with the administrator, and is used to configure the SRA as described below. The management console GUI is Java-based, with all state information stored on the server. It runs as a locally installed Java application in a Java Virtual Machine (JVM**).
- *Management server*: The management server is the platform used for the central management server, where management databases, the server engine, and management application logic reside.

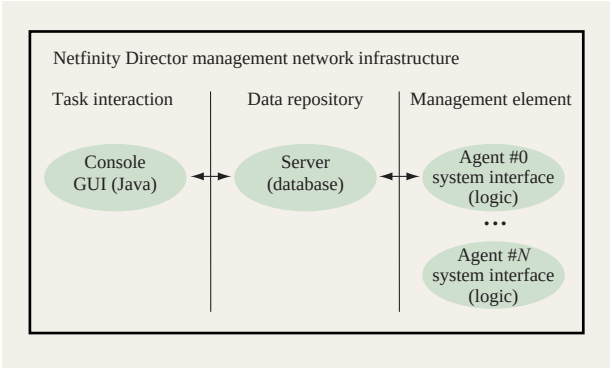

Figure 1

IBM Director framework.

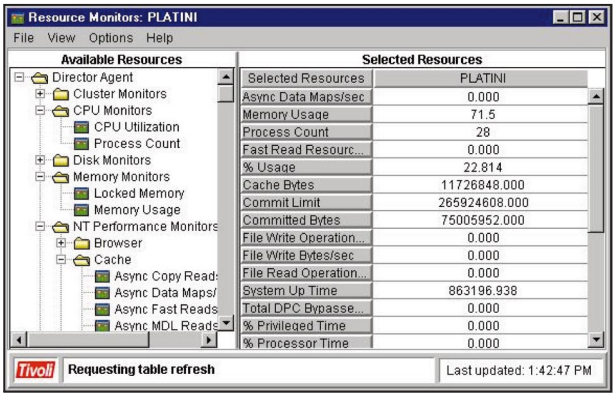

Figure 2

Resource monitors.

- *IBM Director agents*: The agents reside on each managed system (such as an xSeries server) and act as passive, nonintrusive native applications. The SRA task that collects data, predicts resource exhaustion, and generates events runs as an agent.

In addition to the SRA function, IBM Director supports a comprehensive set of tasks for agent nodes. These nodes communicate directly with the IBM Director server, allowing numerous tasks to be performed, of which the following short list is representative:

- *Inventory*: IBM Director discovers new managed systems, collects the appropriate information about these systems, and stores it in the inventory database. It can then be viewed through either a default or a customized view.
- *Resource monitors*: Resource monitors (**Figure 2**) enable the user to view statistics and usage of critical resources on the

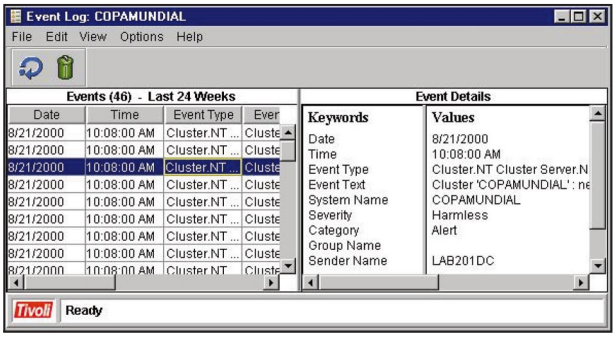

Figure 3

Event management.

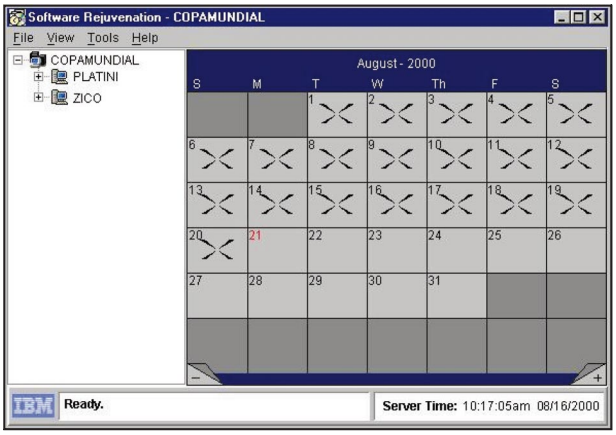

Figure 4

Software rejuvenation calendar.

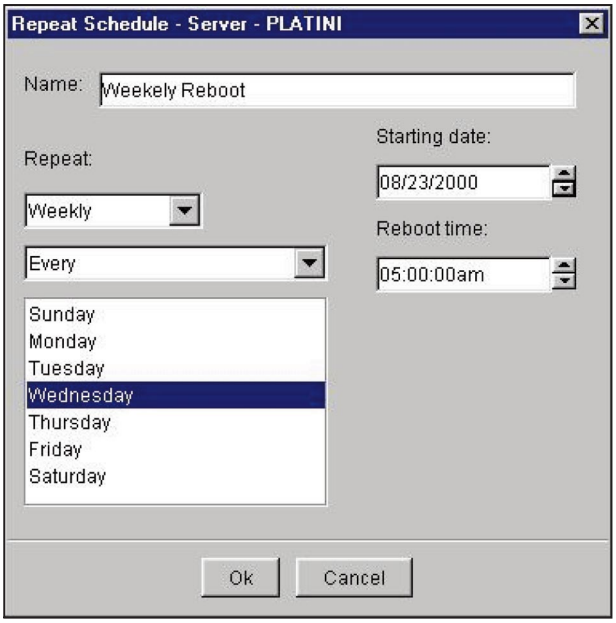

Figure 5

Scheduling rejuvenation.

network. Information can be collected and monitored on attributes such as CPU, disk, memory, and network. SRA is a specialized instance of a resource monitor.

- *Event management*: Event management (**Figure 3**) enables the user to view a log of events that have occurred for a managed system or group of systems and to create event action plans to associate an event with a desired action, such as sending an e-mail, starting a program, logging to a file, or invoking rejuvenation. When the SRA has detected an impending resource exhaustion, it generates events that can be viewed using this functionality.

Software rejuvenation in the IBM Director environment

User interface

Software rejuvenation is presented to the user as a highly stylized means to schedule rejuvenation. The user has at his disposal the building blocks of a schedule and a set of resources. The resources may be scheduled for time-based rejuvenation or configured for exhaustion forecasting.

The SRA user interface is presented to the user in the form of a calendar (**Figure 4**) in which the user conceptually can see and manipulate rejuvenations. To schedule a rejuvenation for a certain day, the user drags and drops a node of the cluster from the left-hand side of the interface (e.g., "platini" and "zico" are servers within cluster "copamundial") onto the day of the week when rejuvenation is desired. A follow-up dialogue (**Figure 5**) then negotiates whether the user wishes the rejuvenation to occur daily, weekly, monthly, or on some other periodic basis, and the time at which the rejuvenation is to occur. The user can designate certain days of the week as invalid, when no rejuvenation can occur, and multiple rejuvenations are prohibited from being scheduled at the same time. Among other rejuvenation options, the user can engage cluster-specific logic designed to ensure that a properly planned failover can occur. As shown in the left-hand side of **Figure 6**, the user can ask the rejuvenation logic to confirm that at least one backup node exists which can handle the failover workload prior to rejuvenating ("check for one"), ensure that all backup nodes can handle the workload ("check for all"), or rejuvenate without checking ("skip check"). If one of the first two

options is selected and a backup node cannot be found, rejuvenation is postponed until the next opportunity.

The proactive option of exhaustion prediction is provided to evoke an advanced and noninteractive scheme for scheduling resources for rejuvenation. The user can set up a stand-alone system or a clustered system for this level of support. In the expert mode of operation for exhaustion prediction, the user can allow an application to schedule rejuvenation automatically without his interaction. The user configures a cluster or a node for prediction capabilities by invoking a simple prediction configuration menu (**Figure 7**), and selecting the notification horizon and the type of action desired when exhaustion is predicted to occur within that horizon. Very few other parameters are configurable by the user.

A notification mechanism, via the built-in IBM Director's event mechanism, allows for notification of any impending exhaustion through either a pop-up or ticker tape. All notifications are done with the IBM Director event-driven notification mechanism. These events are used to drive the scheduling of an automatic rejuvenation, of notifications, and of other user-defined actions (such as running a remote program). The status is reported using the IBM Director-provided event log mechanism.

Agent design

The xSeries Software Rejuvenation Agent is responsible for monitoring the exhaustible parameters, predicting their exhaustion, and providing alerts to the management infrastructure. The data acquisition component of the agent is specific to the operating system. For Windows**, it reads the registry performance counters and collects parameters such as available bytes, committed bytes, nonpaged pool, paged pool, handles, and logical disk utilization. (Interestingly, a memory leak in the system call required to obtain the performance counters required us to design the agent so that it could be rejuvenated.) For Linux, the agent accesses the */proc* directory structure and collects equivalent parameters such as memory utilization, file descriptors, I-nodes, and swap space. All collected parameters are logged on disk. They are also stored in memory in preparation for the curve-fitting and consumption extrapolation.

Prediction algorithms

In the current version of SRA, rejuvenation can be based on elapsed time since the last rejuvenation, or on prediction of impending exhaustion.

When using timed rejuvenation, the user interface of IBM Director is used to schedule and perform rejuvenation at a period specified by the user. A calendar interface allows the user to select when to rejuvenate different nodes of the cluster, and to select "blackout" times during which no rejuvenation is to be allowed.

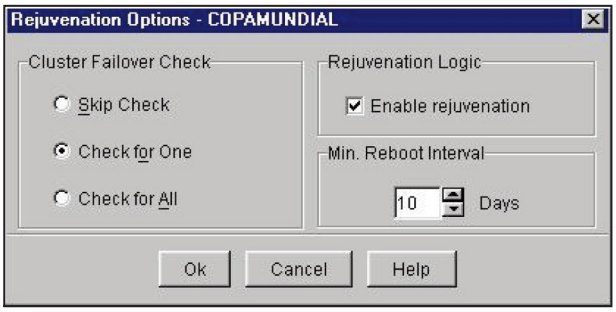

Figure 6

Scheduling options.

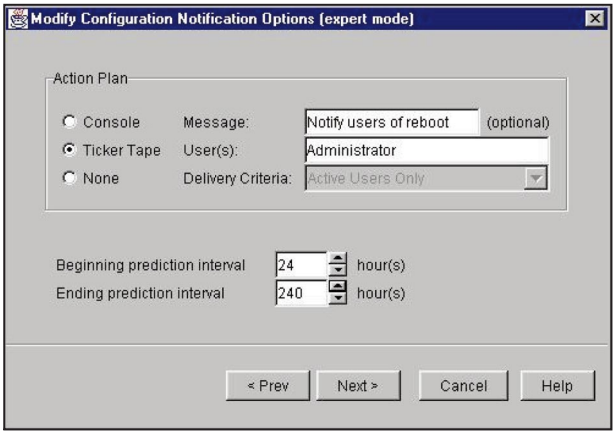

Figure 7

Configuring for prediction.

Although this sounds rather primitive, our analysis (presented below) shows that for typical clusters that undergo aging, system availability can be improved significantly via this technique.

Single-parameter predictive rejuvenation relies on curve-fitting analysis and projection, using recently observed data. The projected data is compared to prespecified upper and lower exhaustion thresholds within a notification time horizon. The user specifies the notification horizon and the desired parameters (some parameters believed to be highly indicative are always monitored by default), and the agent automatically performs the analysis.

The curve-fitting algorithm operates on a sliding window of data spanning a temporal interval which is a fixed fraction (say, 1/3) of the notification horizon. For example, if the user wishes to be informed or have rejuvenation invoked if exhaustion is projected to occur within, say,

three days, the data window is set to one day, and the analysis and extrapolation over the three-day horizon are performed using that one day's worth of data. The sampling interval is selected to provide enough data points within the fitting window to allow the prediction algorithm to adequately smooth the data and select an appropriate prediction function without overfitting the data.

The prediction algorithm fits several types of curves to the data in the fitting window; these curves have been selected for their ability to capture different types of temporal trends. A model-selection criterion is applied to choose the best prediction curve, which is then extrapolated to the user-specified horizon. Several parameters that are indicative of resource exhaustion are monitored and extrapolated independently. If any monitored parameter exceeds the specified minimum or maximum value within the horizon, a request to rejuvenate is sent to the management infrastructure. In most cases, it is also possible to identify the process that is consuming the preponderance of the resource being exhausted, in order to support selective rejuvenation, as described below.

Details of the curve fitting and model selection are given in Appendix A.

Rejuvenation granularity

When an exhaustion has been predicted, the question arises as to what portion of the environment should be rejuvenated. The simple approach to rejuvenation is to perform a node reboot. A single computer node is a distinct and compartmentalized unit of the user's resource environment, and the cluster-management framework within which we are operating should handle such node failures quite adequately. The drawback is that an entire node rejuvenation may be considered too broad in scope, and the time to reboot may be significant. Perhaps only one application on that system, only one of that application's services, or perhaps even a particular subprocess of that service is the cause of the pending exhaustion. The narrower the scope of rejuvenation, the smaller the disruption will be to the user's business objective. From a functional point of view, the operating system provides various APIs to perform rejuvenation at all of these various levels. Whether the resource to be rejuvenated is at the system level, the application level, or even the service level, it is likely that the planned downtime (i.e., the rejuvenation) is always less disruptive than an unplanned node outage.

Because of these considerations, the Windows SRA offers two levels of rejuvenation, depending on the scope of the resource exhaustion (i.e., whether an operating-system-level exhaustion or an application-level exhaustion has been identified). The desired level can be selected using an advanced submenu of the user interface described above.

- *Level 1* is a service-level rejuvenation. Generally, it can be assumed that applications written as a service are written such that a stoppage of that service will save any necessary data (both user and application) and the corresponding restart of that service will bring the application to a state of usability. In such cases, a graceful rejuvenation of that service can be performed.
- *Level 2* is an operating-system-level rejuvenation (i.e., a reboot). The reboot performs a stop for each service on that system and then reboots the operating system. Application failover and recovery in this case are the responsibility of the cluster-management software, which is activated as part of the reboot process.

4. Modeling and analysis

We have developed several availability models to assess the impact of time-based and predictive rejuvenation policies on cluster system availability. These models were developed using stochastic reward nets (SRNs) [28]. SRNs have recently gained much importance and recognition as a useful modeling formalism. They have been used successfully in many applications, since they can easily represent the concurrency, synchronization, sequencing, and multiple resource possession that are characteristics of current computer systems. SRNs also offer a high-level interface for the specification of Markov models. When the state space of Markov models is large, SRNs can be used to generate the underlying Markov chain automatically. SRNs are obtained by specifying reward rates at the net level. We have used SRNs for our modeling and analysis; an informal description of their concepts is given in [29].

System characteristics

We consider an n-node cluster running cluster-management software. Each node has redundant network connections, and the entire cluster contains a redundant storage subsystem. Both hardware and software failures may occur in the cluster, but in this paper we consider only software failures. Failures due to applications, middleware, and operating system are not differentiated, and node failures are assumed to be independent of one another. We also allow common-mode failures which model the failure of the cluster management software, and nonunity failure coverage for a node failure. All failures and repair times are assumed to be exponentially distributed. However, the time-based policies use rejuvenation intervals that are deterministic, and are approximated using an Erlang distribution. Details of these models are given in Appendix B; only the results are presented here.

Table 1 Parameter values.

Transition	*Rate*
λ_1	1/240 /hr
λ_2	1/720 /hr
λ_3	1/30 /min
λ_4	1/4 /hr
λ_5	1/10 /min

Analysis and results

Parameter values such as failure and repair rates used for the models are shown in **Table 1** and explained in Appendix B. Although they fall within the range of existing systems, the values used for this paper should be considered for illustration purposes only. To be able to detect clearly the effect of the rejuvenation, we set the value of the common-mode failure rate (transition T_{cmode}) very close to 0 and the value of the node failure coverage (c_1) very close to 1. Future analyses will explore the effect of rejuvenation policies on availability when these parameters are nonideal. All of the models were solved using the stochastic Petri net package (SPNP) tool [30, 31]. Our SRN model computes the expected unavailability and the expected cost incurred per unit of time. The expected unavailability is computed as the probability that the maximum tolerable number of nodes are inoperative, either because all have failed, or because of some undergoing rejuvenation, with others failing in the meantime. Cost can be incurred because of a node failure, a node rejuvenation, or a system failure (all nodes down). For each case, the cost per unit of time is multiplied by the expected number of tokens in the corresponding places, and the total cost is computed. It is assumed that the cost of rejuvenation is much less than the cost of a node or system failure, since rejuvenation can be done at predetermined or scheduled times (for example, when the system load is low).

We define p_{unav} and c_{fail} to be the steady-state expected unavailability and the expected cost incurred per unit of time, respectively. Over a given time interval T, the expected downtime can be computed as $T \times p_{\text{unav}}$ and the expected cost incurred during that interval as $T \times c_{\text{fail}}$. For concreteness, we rather arbitrarily fix the value of T at 1000 hours. By using these measures, optimal rejuvenation intervals can be obtained for different policies and configurations. The models also help us compare one rejuvenation policy with another. For example, we may not obtain the same rejuvenation interval, which minimizes both unavailability and cost. Therefore, there is a tradeoff involved, and it is up to the user/operator to decide what he/she considers important. One could also optimize the rejuvenation interval based on other performability (combined performance and availability) measures, such as the mean number of nodes operational at a given instant.

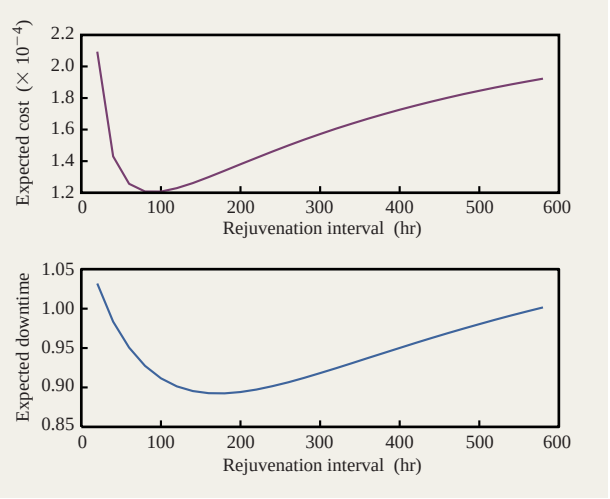

Figure 8

Simple time-based rejuvenation for 8/1 configuration. This plot shows the expected cost and downtime incurred, over a period of 1000 hours, for an eight-node cluster that can tolerate at most one simultaneous node failure.

All of the costs are formulated as costs per unit of time. In our analysis, we fix the value for $cost_{\text{nodefail}}$ at \$5000/hr and vary the ratio $cost_{\text{sysfail}}/cost_{\text{rejuv}}$. The value of $cost_{\text{sysfail}}$ is computed as the number of nodes, n times $cost_{\text{nodefail}}$ (although a total system failure can potentially cost much more than a node failure). Unless mentioned otherwise, we fixed the $cost_{\text{sysfail}}/cost_{\text{rejuv}}$ ratio to be 20.

Finally, we analyze the rejuvenation policies for various cluster configurations. A cluster configuration is denoted by n/m, which means that the cluster has a total of n nodes and can tolerate at most m node failures; i.e., the cluster is considered unavailable if more than m nodes are unavailable.

Results for time-based rejuvenation

Figure 8 shows the plots for an 8/1 cluster (i.e., an eight-node cluster that can tolerate at most one node failure) employing simple time-based rejuvenation. The upper and lower plots show the expected cost incurred and the expected downtime (in hours) respectively in a given time interval, versus rejuvenation interval in hours. When evaluating rejuvenation, we typically see J-shaped curves, with a region of high unavailability when the rejuvenation interval is very small (the system is always rejuvenating, so any other node failure while rejuvenating results in a system outage) and another region of high unavailability when the rejuvenation interval is very large (the system essentially never rejuvenates, and the effects of aging have their full impact). Between these extrema lies a rejuvenation interval that results in the minimum

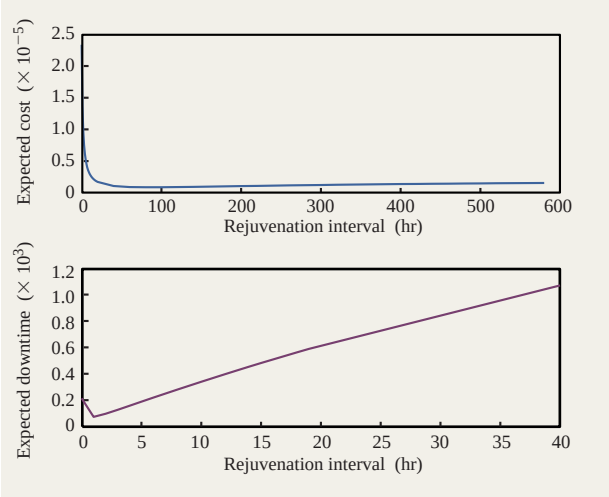

Figure 9

Simple time-based rejuvenation for 8/2 configuration.

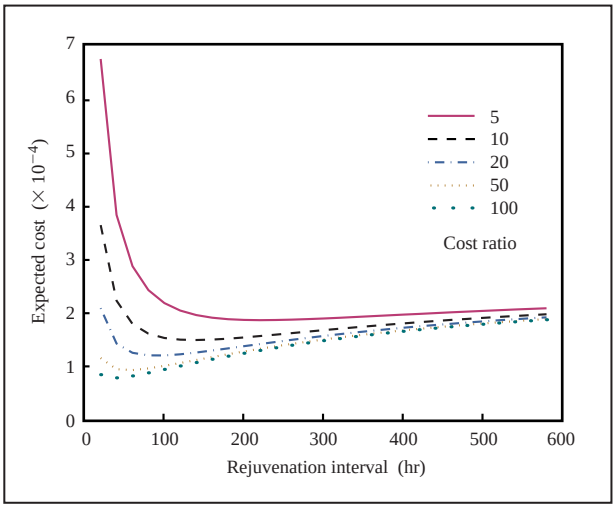

Figure 11

Effect of node failure/node rejuvenation cost ratio for 8/1 configuration employing simple time-based rejuvenation.

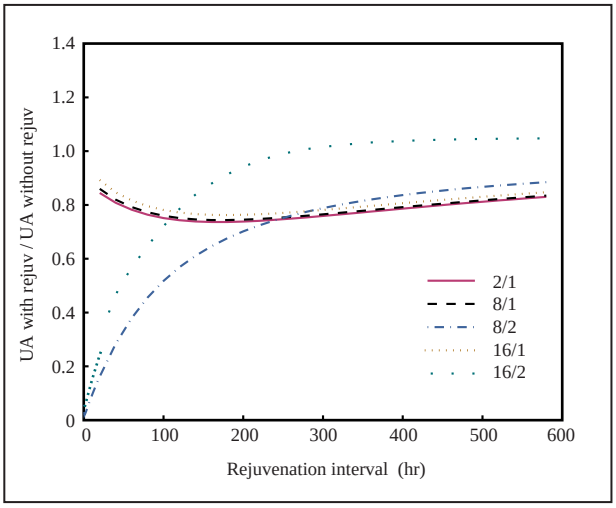

Figure 10

Unavailability improvement for various configurations employing simple time-based rejuvenation.

downtime and cost. For the 8/1 configuration, the minimum for the cost occurs at 100 hours, and the minimum for the downtime occurs at 180 hours. Thus, if one wishes to invoke timed rejuvenation, a compromise has to be made regarding the rejuvenation interval based on whether the expected downtime or the expected cost incurred is more important.

Figure 9 shows similar plots for an 8/2 configuration. In this case, a shallow minimum for the cost occurs at 80 hours, and a sharp minimum for the downtime occurs at one hour. Here we take advantage of the fact that the system can tolerate up to two node failures at the same time, and therefore we can afford to rejuvenate more frequently. Hence, the optimal cost and downtime are also substantially reduced.

A significant figure of merit for a rejuvenated system is the unavailability improvement due to rejuvenation, which we define to be the ratio between the expected steady-state unavailability with rejuvenation and the unavailability without rejuvenation. A plot of unavailability improvement versus rejuvenation interval is shown for different configurations in **Figure 10**. In general, there is a much larger improvement in unavailability for the X/2 configurations (that is, X-node clusters that can tolerate two failures before becoming unavailable) than that for X/1 configurations. For the 8/2 and 16/2 configurations, the optimal rejuvenation interval occurs very close to zero. But as the rejuvenation interval increases, they deteriorate rapidly, and the improvement becomes less than that for the X/1 configuration.

Next, we studied the effect of the $cost_{sysfail}/cost_{rejuv}$ ratio on the rejuvenation interval. This was done for the 8/1 configuration and is shown in **Figure 11**. As the cost ratio is increased (i.e., rejuvenation becomes cheaper and cheaper relative to a system outage), the optimum value of the rejuvenation interval d decreases and the overall expected cost also decreases. As $d \to \infty$, there is no rejuvenation, so the total cost ceases to be a function of the rejuvenation cost. In this case, all of the different cases approach the same value.

Results for prediction-based rejuvenation

In our model of prediction-based rejuvenation, a node is rejuvenated as soon as it enters a state in which it has a high failure rate, instead of after a specified inter-rejuvenation time interval. To model imperfect prediction, our analysis includes a "prediction coverage" factor, which is the probability that the failure predictor successfully detects that a node is likely to fail, given that it has just entered the high-failure-rate state. (We do not model the case in which the failure predictor emits a false alarm and triggers an unneeded rejuvenation, although this is straightforward. We think that this phenomenon has a minor effect on unavailability for reasonable false-alarm probabilities.) The plot in **Figure 12** shows the transient values of expected downtime over a 5000-hour operational interval, for the 8/1 configuration. The steady-state values, achieved after approximately 2000 hours of operation, are found toward the right of the chart. The graphs show the values for different values of prediction coverages. As we would expect, the higher the prediction coverage, the lower the expected downtime. At 100% prediction coverage, the data lies along the *x*-axis and is practically invisible. We compare these steady-state values with the time-based policy in the following section.

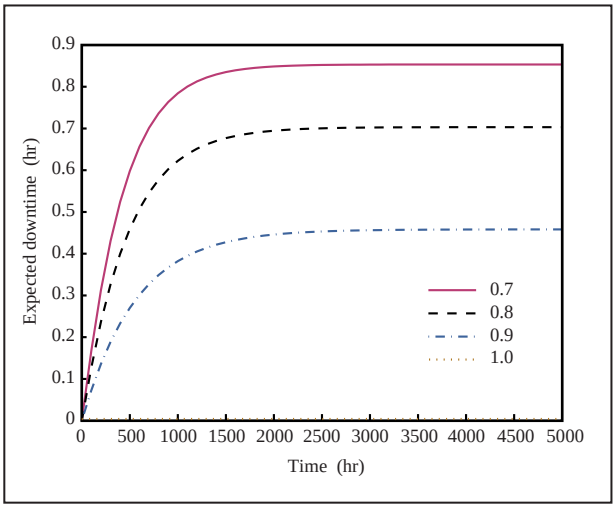

Figure 12

Prediction-based rejuvenation for 8/1 configuration for different prediction coverage values.

Summary of analytical results

The data in **Table 2** shows the ratio of downtime with rejuvenation to downtime without rejuvenation, for time-based and predictive rejuvenation. For time-based rejuvenation, a rejuvenation interval is used which minimizes downtime, which was approximately 100 hours for the one-spare clusters, and approximately one hour for the two-spare clusters. For predictive rejuvenation, it is assumed that aging can be successfully predicted 90% of the time. If we assume that aging can be predicted 100% of the time, downtime drops to practically zero, but we do not feel that perfect predictive coverage is achievable in practice.

The analysis results lead to some interesting conclusions. First, for systems that have one spare, and regardless of the number of nodes in the cluster, time-based rejuvenation can reduce downtime by about 25% compared with no rejuvenation. Predictive rejuvenation does somewhat better, reducing downtime by about 60% compared with no rejuvenation. However, when the system can tolerate more than one failure at a time, downtime is reduced by 98% to 95% via frequent time-based rejuvenation, compared to a mere 85% for predictive rejuvenation. We believe that this is because, with high-frequency time-based rejuvenation, a node spends very little time in the "failure-prone" state before it is rejuvenated back to the low-failure-rate state. For predictive rejuvenation with only 90% coverage, a node whose aging has gone undetected remains in the "failure-prone" state until it actually fails; as we currently model it, there is no second chance to be rejuvenated. Therefore, there is a higher probability that one node can have failed (or is being rejuvenated), a second is being rejuvenated (or has failed), and a third node fails, having escaped aging detection. We point out that, at the high levels of availability predicted by our model for the two-spare clusters, it is likely that nonideal effects such as single points of hardware and software failure and nonunity failover coverage would dominate the effects of rejuvenation we are predicting here. However, this is probably not the case for the one-spare configurations.

Table 2 Ratio of downtime with rejuvenation to downtime without rejuvenation.

Rejuvenation policy	*Configuration*				
	2/1	*8/1*	*16/1*	*8/2*	*16/2*
Time-based (optimal rejuvenation interval)	0.74	0.74	0.76	0.02	0.05
Prediction-based (90% prediction coverage)	0.38	0.38	0.39	0.15	0.15

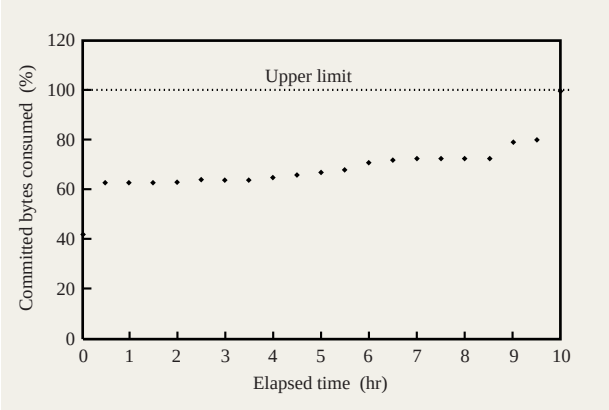

Figure 13

Committed bytes versus time (in hours) for a large database application.

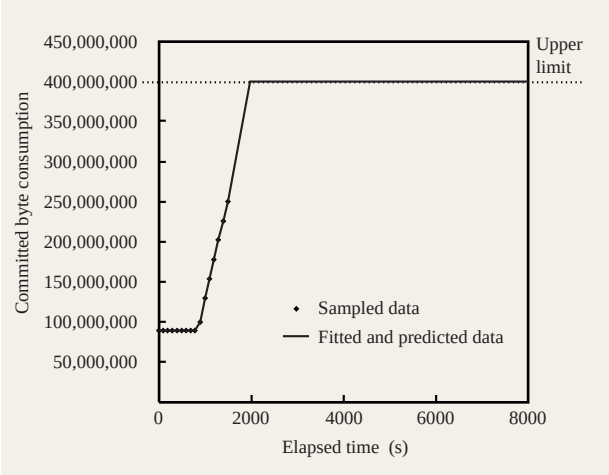

Figure 14

Committed byte consumption versus time (in seconds) for a leaky application in Windows, with exhaustion prediction.

5. Experimental results

Empirical measurement of resource exhaustion

In the process of developing and testing the SRA, it was necessary to confirm the existence of aging, understand which parameters were most likely to age, and learn how to measure and predict resource exhaustion for certain applications of commercial interest to IBM. For this reason, several important applications were put under high workload and their aging characteristics were measured. The chart in **Figure 13** shows typical results, in which one such large-database application running on the Windows operating system consumed 100% of the committed bytes (e.g., page-file bytes plus physical memory) over a ten-hour interval, beginning at an initial utilization level of 45%. Other resources were consumed, but the committed bytes resource was finished off first. (In the interest of accelerating the test, the database was presented with a workload that was much higher than would be encountered in a well-run production environment. Anecdotal data indicates that in a real-world environment, exhaustion for this application requires more than a week.) For this particular application, once the committed bytes resource was consumed, the database was unable to proceed, and usually panicked or hung. Once the resource was consumed and the panic had occurred, a full system reboot and database reconstruction was required to bring this particular database back on line. An interesting result of our testing regime was the discovery that of the applications that did show aging, the same kinds of resources were usually consumed. This implies that the ability to predict or detect a relatively small set of failure signatures is probably sufficient to cover an adequately large set of applications, for practical purposes.

"Bad boys"

Large applications are difficult and time-consuming to set up and test. In the interest of accelerating the test process and fine-tuning and testing our algorithms, we also generated several applications that were prodigious consumers of disparate resources. **Figure 14** shows how one such "bad boy" application, again running on Windows, consumes committed bytes, starting at 800 seconds into the experiment. (We have other bad boys that consume processes, threads, nonpaged and paged pool memory, semaphores, handles, etc.) For purposes of testing, the notification horizon was set to 7200 seconds, and the sampling rate and resource consumption rates were scaled appropriately. In practice, much larger time frames would be used. This particular figure also shows how the piecewise regression of the SRA predictive agent compensates for the startup transient resource consumption, and predicts the time to exhaustion for this scenario. The resource-exhaustion upper limit of about 400 MB is shown as a dashed line at the top of the plot. **Figure 15** shows how another bad boy application that continually opens files but forgets to close them consumes I-nodes on the Linux operating system. Again, testing is accelerated, and the notification horizon is set to 3600 seconds; the lower limit is somewhat arbitrarily set to 500 I-nodes.

6. Conclusions and extensions

We have developed, analyzed, and implemented a framework for detection, prediction, and proactive management of software aging. This technology is

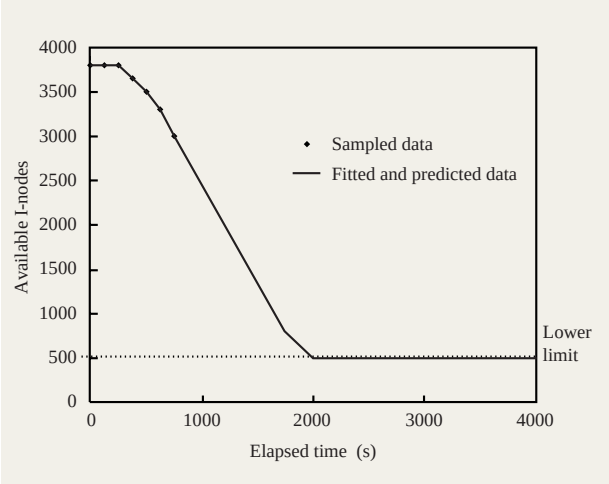

Figure 15

I-node consumption versus time (in seconds) for a leaky application in Linux, with exhaustion prediction.

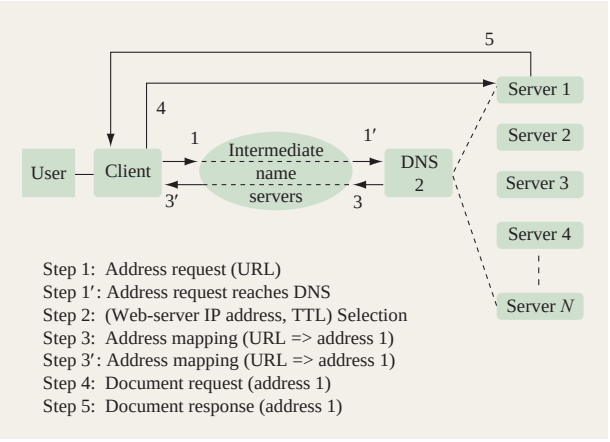

Figure 16

Example configuration of round-robin DNS.

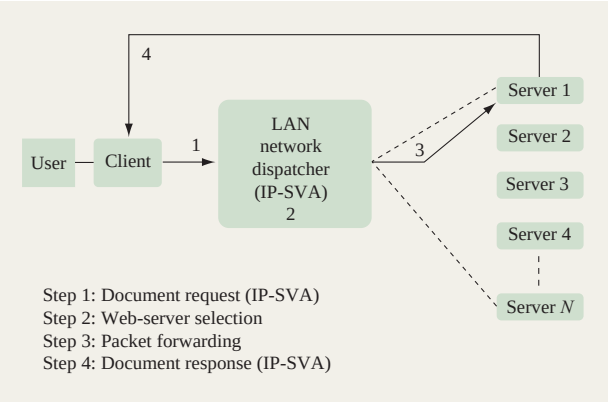

Figure 17

Network dispatcher for a LAN implementation.

applicable to a wide range of operating environments, and has been implemented in the xSeries Software Rejuvenation Agent. It has been commercially available on xSeries servers since the end of 2000. Our cost and availability models indicate that rejuvenation significantly improves cluster system availability and reduces downtime cost.

We have considered several extensions and other applications of software failure prediction and state rejuvenation. This section outlines a few of our thoughts in this area.

IP dispatching

As part of a web-hosting framework, an IP-dispatching or load-balancing component is often present to provide scalability, availability, and load-balancing capabilities for TCP/IP applications. Early implementations consisted of a domain name server (DNS) that would translate host names into IP addresses for corresponding servers, so that IP requests are routed in a round-robin fashion to a pool of servers. This approach is often called round-robin DNS; an example configuration is shown in **Figure 16**. Note that IP dispatching can be considered a specialized form of clustering, such that one or more nodes execute the dispatching components and the remaining nodes execute web-serving applications.

More advanced approaches are now available, such as the IBM Secureway Network Dispatcher [32]. This product provides enhanced IP-level load-balancing mechanisms and content-based routing, as well as improved management and availability functions. **Figure 17** illustrates the network dispatcher operations for a LAN implementation. As part of load balancing, the dispatcher's scheduling policy is dynamically based on each server's load and availability. This is partly accomplished by having each server send periodic utilization information to the dispatcher. This utilization information can easily be augmented by health information in the form of time until resource exhaustion, degree of resource exhaustion or, in its simplest form, time remaining until a timed rejuvenation. This health information can be sent to the dispatcher in order to schedule actions for individual servers. The scheduling of these actions can also take into account aggregate loading of the web host.

Solution center

One of the core pieces of the software rejuvenation technology is a monitoring tool that has the ability to detect and identify misbehaving software components. In addition to this technology's application in production clustering applications, it also has use as a test tool to help make software more reliable. This type of tool could be useful in a system integration facility such as an IBM solution center, where software of disparate quality is integrated onto a common platform. (We have first-hand experience of this. The ability of the agent to detect and pinpoint resource consumption was useful in developing and testing the agent itself, when some of the system calls used to monitor resource consumption caused resource leaks themselves.)

Discrimination between hardware and software faults

When a system crashes, it can be difficult to determine whether the crash was due to hardware or software, especially if a subsequent system reboot is successful. Frequently, hardware is identified as the culprit when it is associated with a certain number of outages, perhaps because the service technician has to do *something* that is perceived as solving the customer's problem. Consequently, nonfaulty hardware is often returned to IBM under a service contract, at a significant cost to the corporation. We think that a variant of the software-monitoring agent we have developed can provide valuable clues to the user or technician as to whether the crash was due to a software problem it is capable of detecting, possibly reducing the number of no-fault-found hardware returns and IBM service costs. It could also be expanded to monitor hardware errors to improve its diagnostic resolution, since it is fairly well known that permanent hardware failures are often preceded by an increasing rate of occurrence of transient hardware failures.

Adaptive and multiparameter predictive capabilities

The current version of SRA was developed on the basis of a preconceived notion of the types of resources that can be exhausted and their exhaustion thresholds, for a given set of operating systems. During development, this hypothesis was supported by a testing and validation effort for several applications of importance to IBM, and was deemed valid. Consequently, the SRA is capable of monitoring and predicting the exhaustion of any single parameter that is on a fairly static initialization list, using a flexible curve-fitting methodology. We think this capability is adequate for a sufficiently large number of applications to make the product useful. However, it is not known in general which parameters may be exhausted or enter critical regions in all scenarios, nor what their exhaustion thresholds are. An outage may occur only when a combination of parameters reaches critical regions, or it may be the case that a given parameter or combination of parameters does not have to be at extrema to constitute a hazardous situation. For example, we noticed during our testing of a web-serving application that just prior to the outage, the committed bytes, nonpaged-pool bytes, and available memory approached known exhaustion limits, as expected. However, other parameters also repeatedly exhibited unusual and indicative behavior just prior to the outage: For example, the variance of the paging rate and the number of nonpaged-pool allocations skyrocketed, although these do not seem to be outage precursors on their own. Therefore, we think that a truly general outage predictor must have the capability to characterize possibly complex multiparametric conditions prevalent just prior to outages that occur in a given application, store this characterization in an analytically tractable form, and develop the ability to predict a system's subsequent approach to that region in multiparameter space.

Availability modeling and analysis

As part of the future work in modeling and analysis, we could introduce new cluster-system characteristics and failure behavior and improve the models. We could analyze these models for many more different configurations than have been explored in this paper. Other new and interesting performability measures could be introduced for the analyses. To obtain more accurate results using the SRN models, we could use the theory of Markov regenerative processes (MRGP). Another possible solution method for general non-Markovian models is discrete-event simulation. The models could be improved to consider hardware components and failures and discuss their impact on system availability and performance. New rejuvenation policies could be formulated based not just on time, but also on the load of the system (instantaneous or cumulative) [24].

Appendix A: Exhaustion-prediction algorithms

This appendix is devoted to the description of the extrapolation and model-selection algorithms for predictive rejuvenation. The steps of the prediction procedure are as follows:

- Preprocessing the data.
- Fitting several models to the preprocessed data.
- Selecting the best model.
- Forecasting the data behavior with the selected model.

Let $X_1, \cdots, X_m$ denote the observations in the training (fitting) window, and let $T_1, \cdots, T_m$ be the corresponding sampling times. Using a median filter, the data is preprocessed to produce n medians $Y_1, \cdots, Y_n$, which we associate with "sampling times" $t_1, \cdots, t_n$. In particular,

let the median filter operate on k samples (say $k = 5$); then, assuming $m = n \times k$, $Y_1 = \text{median}(X_1, \cdots, X_k)$, $Y_2 = \text{median}(X_{k+1}, \cdots, X_{2k})$, etc., and the sampling times of the medians are defined as $t_1 = \text{median}(T_1, \cdots, T_k)$, $t_2 = \text{median}(T_{k+1}, \cdots, T_{2k})$, etc. Median filtering produces a smoothed version of the signal and is quite robust with respect to the presence of spikes in the time series.

Our goal is to select a parsimonious model that adequately describes the data. To that end, we consider a relatively small number of model classes, pick the best model from each class (i.e., fit the model to the data), and then, from among those, select the "best" overall model.

To pick the best model from a parametric class $\boldsymbol{M}$ (for example, $\boldsymbol{M}$ could be the set of linear models $Y_i = at_i + b + \epsilon_i$, where the parameters are a and b), we select the parameter values that optimize a goodness-of-fit criterion. In particular, we minimize the residual sum of squares

$$RSS_{\boldsymbol{M}} = \sum_{i=1}^{m} (\hat{Y}_i - Y_i)^2,$$

where $\hat{Y}_i$ is the value produced by the model at time t_i. Minimizing the residual sum of squares for the classes of models used in the system is computationally efficient.

To ensure a rich enough selection, we use the following classes of fitting models:

- *Simple linear regression*: $Y_k = at_k + b + \epsilon_k$. The number of parameters is $p = 2$.
- *Linear regression with h breakpoints* (where we use $h = 1$, as in **Figure 18**, or $h = 2$, as in **Figure 19**): Given a set of $h + 2$ time instants $t_1 = c_0 < c_1 < \cdots < c_h < c_{h+1} = t_m$ (i.e., the k breakpoints and the endpoints of the analysis window), $Y_k = a_j t_k + b_j + \epsilon_k$ if $c_{j-1} \le t_k \le c_j$, where we constrain the coefficients a_j and b_j so that the overall fitted piecewise-linear curve is continuous. Given the time instants $\{c_i\}$, the fitting problem can be formulated as a linear regression and is easily solved. The problem, however, is not convex in the selection of the breakpoints $c_1, \cdots, c_k$. To keep the model-fitting problem computationally efficient, we constrain the times of the breakpoints to occur at a small number B of possible locations (say, $B = 10$) through the fitting window. To avoid overfitting of the data when a spike or a jump occurs right at the end of the fitting window, making the extrapolation unreliable, we also do not allow breakpoints near the end of the window. The selection algorithm operates by fitting a model to the data for each valid selection of breakpoints, and choosing the model with the minimum RSS. Note that if $h = 1$, the number of parameters p equals 4 (for a given breakpoint, the piecewise-linear curve is defined by three points, so add one more parameter for the location of the breakpoint). Similarly, if $h = 2$, $p = 6$.
- *Linear regression on the logarithm of the data*: $\log(Y_k) = at_k + b + \epsilon_k$. The number of parameters is $p = 2$.

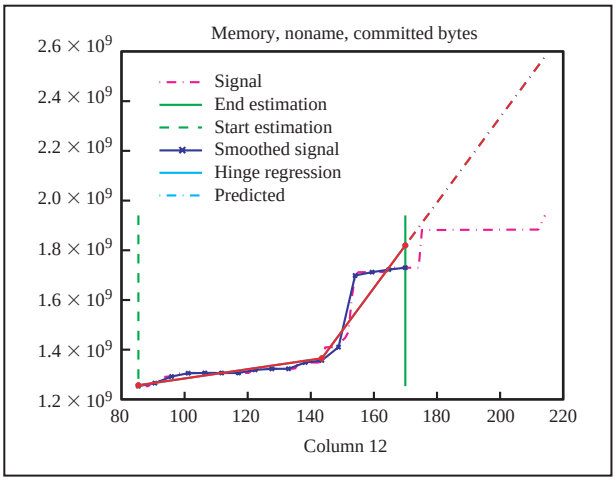

Figure 18

Result of fitting the smoothed data (solid curves connecting × marks) with a piecewise linear regression with a single split. The vertical lines denote the limits of the fitting window. The regression fit, within the fitting window, is denoted by a solid curve, and the prediction is its dash–dot right continuation. The original unsmoothed data is also shown as a dash–dot curve.

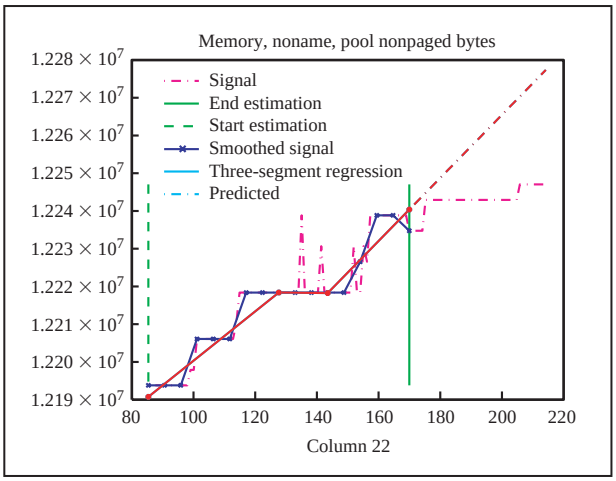

Figure 19

Result of fitting the smoothed data (solid curves connecting × marks) with a piecewise linear regression with two hinge points. The vertical lines denote the limits of the fitting window. The regression fit, within the fitting window, is denoted by a solid curve, and the prediction is its dash–dot right continuation. The original unsmoothed data is also shown as a dash–dot curve.

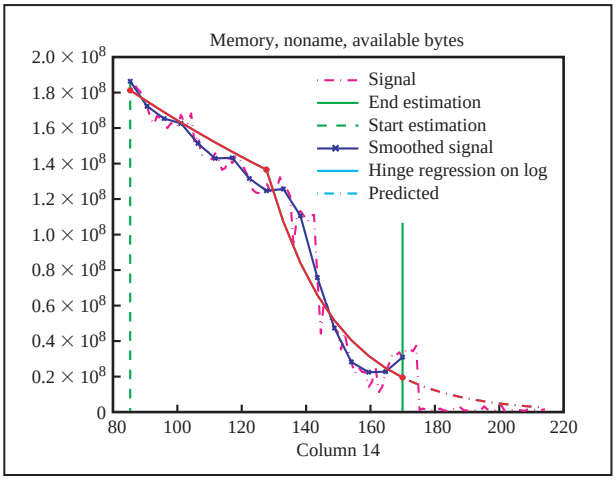

Figure 20

Result of fitting the logarithm of the smoothed data (solid curves connecting × marks) with a piecewise linear regression with a single hinge point. The vertical lines denote the limits of the fitting window. The regression fit, within the fitting window, is denoted by a solid curve, and the prediction is its dash–dot right continuation. The original unsmoothed data is also shown as a dash–dot curve.

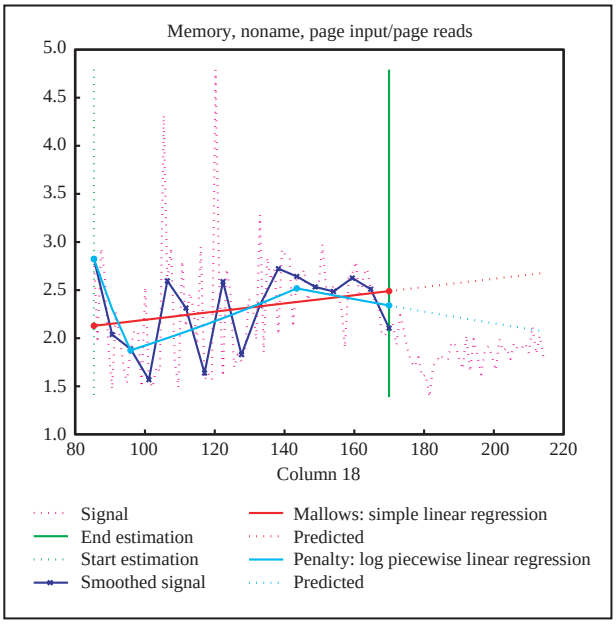

Figure 21

Instance in which the selection criteria pick different models. The criterion based on Mallows statistics selects a simple linear regression, while the criterion based on multiplicative penalty (1 + δ per degree of freedom) fits piecewise regression with two breakpoints to the log of the data.

- *Piecewise-linear regression with k breakpoints on the logarithm of the data*: $\log(Y_k) = a_j t_k + b_j + \epsilon_k$ if $c_{j-1} \le t_k \le c_j$. If $h = 1$, as in **Figure 20**, $p = 4$, and if $h = 2$, $p = 6$.

In choosing the "best" overall model, we wish to avoid overfitting the data. In classical statistics, model selection methods, such as Akaike's information criterion (AIC) [33] or Mallows' C_p index [34], combine a goodness-of-fit measure with a penalty that grows with the order (number of parameters) of the model. Mallows' C_p method is particularly simple to apply. Compute

$$C_p = RSS_M/s^2 - (m - 2p),$$

where p is the number of parameters in the model, s^2 is the residual mean-square error from the largest model (which is assumed to be a good estimate for $\mathrm{Var}[\epsilon_i]$), and m is the number of samples in the smoothed signal. The model with the smallest value of C_p is chosen.

In addition to Mallows' C_p, we use a similar heuristic approach, which is less dependent on distributional assumptions and has several characteristics tailored to the situation at hand. The idea is that, while models are trained on the entire data in the window, the selection criterion should give greater weight to how well each model fits the most recent data. Then define the weighted residual sum of squares as follows:

$$WRSS_M = \sum_{i=1}^{m} w_i(\hat{Y}_i - Y_i)^2,$$

where w_i is the weight assigned to the fit of sample i; for example, $w_i = \rho^{m-i}$ gives decreasing weight to observations further in the past. If $\rho = 0.95$, the error in fitting the most recent observation is roughly three times more important than the error in fitting the 20th more recent observation. Computing the appropriate exact modification to the C_p index is difficult in this situation, so we adopt an intuitive heuristic which does not consider a class of models with an additional parameter unless it reduces the $WRSS_M$ by at least a certain percentage. Specifically, if p_0 is the minimum number of parameters used (with linear regression, two parameters are estimated, hence $p_0 = 2$) and if $WRSS_M(p)$ is the weighted residual sum of squares, the selection criterion picks the model with the minimum W_p, where

$$W_p = WRSS_M(p) \times (1 + \delta)^{p-p_0}.$$

We heuristically select δ to be 0.10. Then, each added model parameter must decrease $WRSS_M(p)$ by at least 10%.

Thus, in each window, six possible model families are considered (linear, linear with one breakpoint, linear with

Table B1 Parameter values.

Transition	*Rate*
λ_1	1/240 /hr
λ_2	1/720 /hr
λ_3	1/30 /min
λ_4	1/4 /hr
λ_5	1/10 /min

two breakpoints, log linear, log linear with one breakpoint, and log linear with two breakpoints), and either the model with minimum C_p or that with minimum W_p is chosen. Experiments show that when no breakpoints are allowed in the most recent part of the fitting window, the C_p and W_p criteria usually yield the same results. Only rarely are different models selected: In particular, the criteria tend to disagree when no trend is clearly discernible, and the data looks very noisy (**Figure 21**).

Prediction of the future behavior of the data is then performed by extrapolating the selected model. Note that each of the six model classes can be cast as a linear regression. Therefore, it is also possible to project approximate confidence limits out into the future. These intervals have the following desirable properties:

- They get wider the further into the future the projection extends.
- Noisier data results in wider intervals.
- The piecewise-linear or piecewise-log-linear models become wider the closer the last breakpoint is to the present.
- The predicted curve is monotone into the future.

Roughly speaking, the computational requirement per monitored quantity is proportional to $m \times 6 \times B^2/2$, where m is the number of medians, B is the number of allowed breakpoint locations, and the factor 6 comes from the six model classes.

Appendix B: Analysis using stochastic reward nets

This appendix details our approach of calculating cluster system unavailability and cost using stochastic reward nets.

Analysis parameters

Table B1 shows the main parameter values that are used in the models. λ_1 is the rate at which a node transitions from a nonfailure-prone state to the failure-prone state, and λ_2 is the rate at which a node fails once it has entered the failure-prone state. λ_3 is the rate at which a node is repaired when it has suffered an unplanned node failure, λ_4 is the rate at which a system is repaired after it has suffered an unplanned system outage, and λ_5 is the rate at which a node is rejuvenated once the need to rejuvenate has been established.

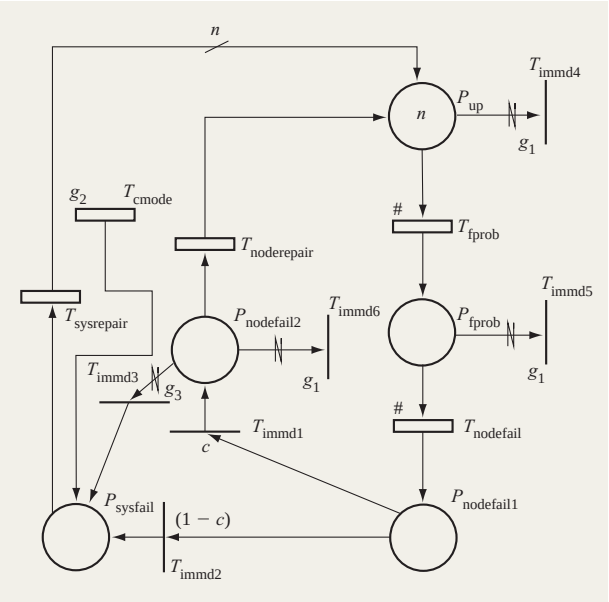

Figure 22

Basic cluster system.

Basic cluster system

Figure 22 shows the basic model of our cluster system. The cluster consists of n nodes. Initially, all of the nodes are in a "robust" working state, indicated by tokens in place P_{up}, in which the probability of node failure is zero. As time progresses, each node eventually transits to a "failure-probable" state (place P_{fprob}) through the transition T_{fprob}. The nodes are still operational in this state but can fail (transit to place $P_{nodefail1}$ with a nonzero probability). If a node crashes, it can recover with a probability c through the transition $T_{noderepair}$. In this case, the node goes back to the place P_{up} which represents the clean state of the node. The node recovery can fail with a probability $(1 - c)$, leading to a system failure (all n nodes are down). Place $P_{sysfail}$ represents this system-level failure state.

Thus, the time to failure for the node starting in the robust state P_{up} has a hypoexponential distribution [35]. Since this is an increasing-failure-rate distribution, it models software aging. From a full system outage, the system can be repaired through the transition $T_{sysrepair}$ and all the n nodes return to place P_{up}. Many applications may require a minimum number of cluster nodes to be up

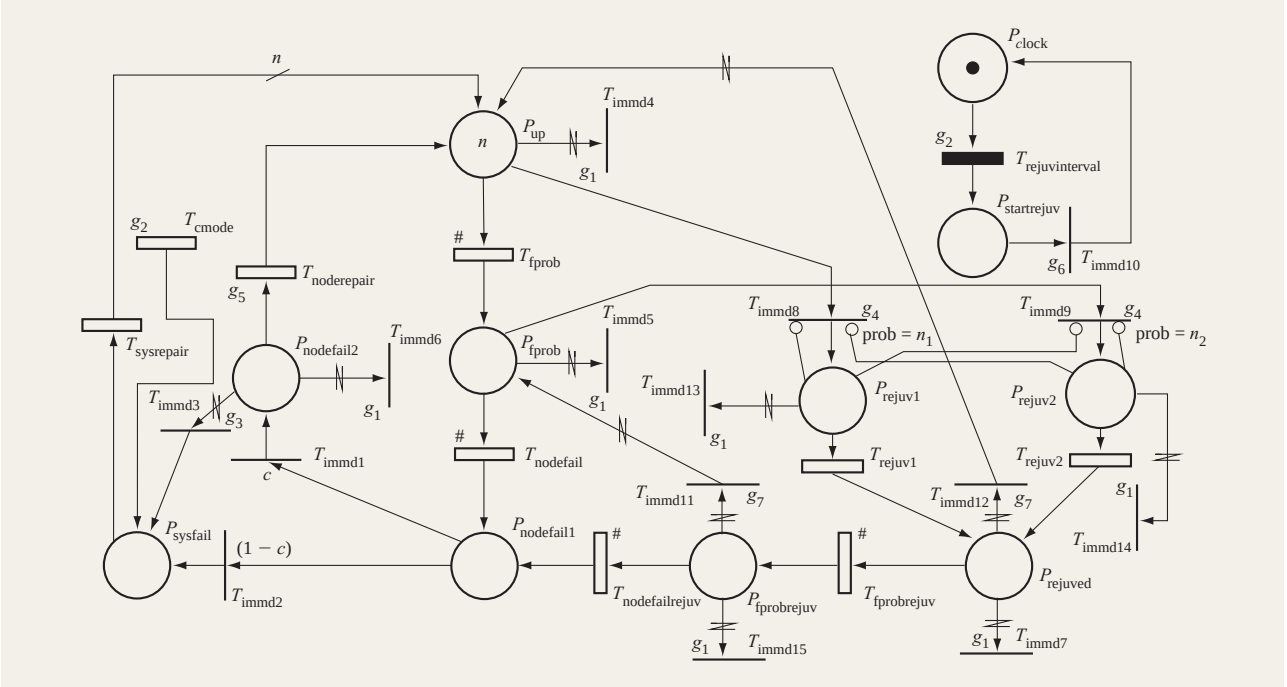

Figure 23

Cluster system employing simple time-based rejuvenation.

for the service to be available. In these cases, the system is considered down when there are a ($a \leq n$) individual node failures. This is modeled by using the guard function g_3, which checks for this condition. In addition to individual node failures, there is also a common-mode failure, which takes place when transition T_{cmode} fires. The common-mode failure causes all nodes that are operational at that instant to be down. The transition representing this failure is enabled at all times the system is in the up state (guard function g_2). When there is a system crash (i.e., there is a token in place $P_{sysfail}$), all of the remaining tokens in places P_{up}, P_{fprob}, and $P_{nodefail2}$ are drained. This is accomplished by defining a guard function g_1 which enables the immediate transitions T_{immd4}, T_{immd5}, and T_{immd6} if there is a token in place $P_{sysfail}$.

Simple time-based rejuvenation

The SRN model for a simple time-based rejuvenation policy for a cluster system is shown in **Figure 23**. In this policy, rejuvenation is done simultaneously for all of the operational nodes in the cluster, at periodic deterministic intervals. The rejuvenation interval is determined by using a clock. Initially, there is a token in the place P_{clock}. The deterministic transition $T_{rejuvinterval}$ fires every d time units and deposits a token in place $P_{startrejuv}$. Rejuvenation begins only when there is a token in this place. If there is a token in place $P_{startrejuv}$ and there are nodes to be rejuvenated (i.e., there are tokens in places P_{up} or P_{fprob}), immediate transitions T_{immd8} and T_{immd9} are respectively enabled. Only one node can be rejuvenated at a time, and nodes are allowed to fail when another node is being rejuvenated. To model this, only one token is transferred from places P_{up} and P_{fprob}. The probability of selecting a token from them is directly proportional to the number of tokens in each. Weight functions n_1 (the number of tokens in P_{up}) and n_2 (the number of tokens in P_{fprob}) ensure this. Since only one node can be rejuvenated at a time, a token is deposited either in place P_{rejuv1} (from place P_{up}) or in place P_{rejuv2} (from place P_{fprob}). Inhibitor arcs from these places to transitions T_{immd8} and T_{immd9} prevent more than one token from coming in. Transitions T_{rejuv1} and T_{rejuv2} are the transitions for rejuvenation from places P_{rejuv1} and P_{rejuv2}, respectively. After a node has been rejuvenated, it goes back to the "robust" working state, represented by place $P_{rejuved}$. This is a duplicate place for P_{up} in order to distinguish the nodes which are waiting to be rejuvenated from the nodes which have already been rejuvenated. A node, after rejuvenation, is then allowed to fail with the same rates as before rejuvenation, even when another node is being rejuvenated. Hence, duplicate temporary places

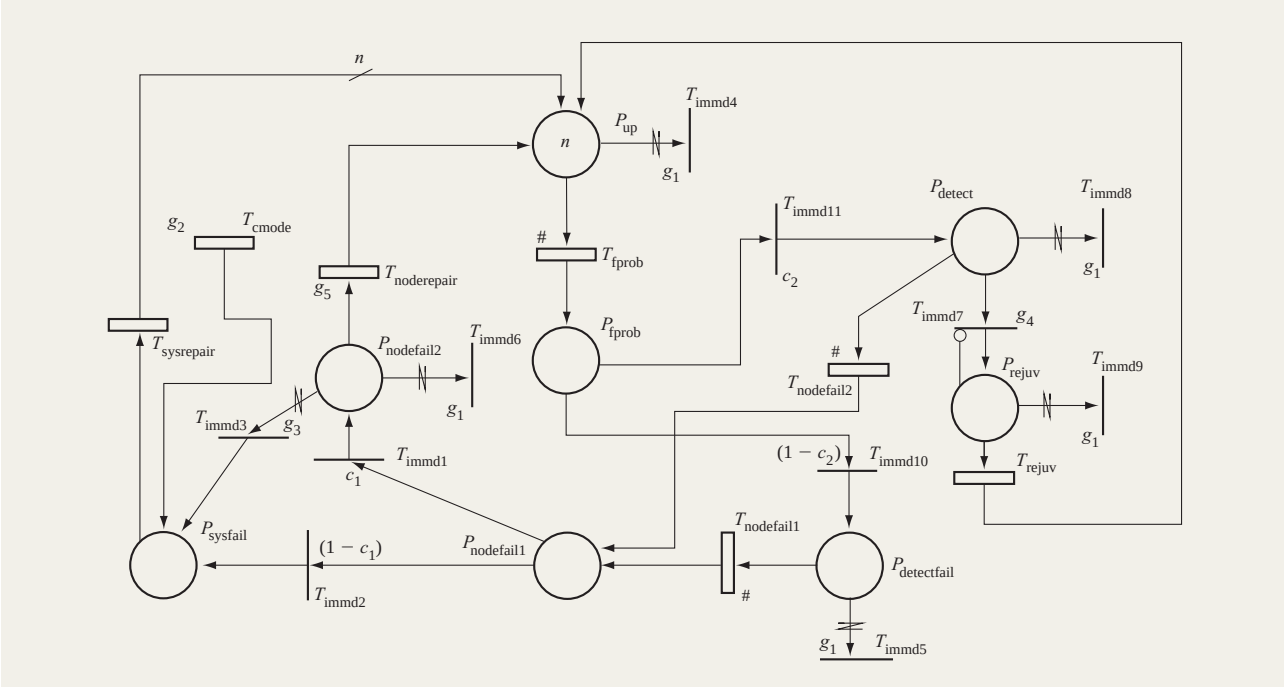

Figure 24

Cluster system employing prediction-based rejuvenation.

for P_{up} and P_{fprob} are needed. The transitions $T_{fprobrejuv}$ and $T_{nodefailrejuv}$ have the same rates as transitions T_{fprob} and $T_{nodefail}$. Node repair (transition $T_{noderepair}$) is disabled during rejuvenation. Rejuvenation is complete when the sum of nodes in places $P_{rejuved}$, $P_{fprobrejuv}$, and $P_{nodefail2}$ is equal to the total number of nodes, n, in the cluster system. The immediate transition T_{immd10} is fired when rejuvenation is complete or when the entire system fails during rejuvenation. When the entire system fails during rejuvenation, tokens are drained from all places in which it is possible to have tokens. Transition T_{immd10} also fires when, just before rejuvenation starts, there are $a - 1$ tokens in place $P_{nodefail2}$. Since a individual node failures leads to a system crash, and rejuvenation is considered a "down" state, rejuvenation does not take place if there are $a - 1$ individual node failures in the system. In this case, the clock simply begins the countdown again. The guard function g_6 checks for these three conditions. The clock is disabled when the system is undergoing repair (when there is a token in place $P_{sysfail}$).

We have approximated the deterministic clock using an r-stage Erlang distribution [36]. If the rejuvenation interval is d time units, each of the r Erlang stages is exponentially distributed with mean d/r. The mean of this Erlang distribution is thus d.

Prediction-based rejuvenation

The SRN model for a prediction-based rejuvenation policy for a cluster system is shown in **Figure 24**. The failure-repair characteristics of the system are the same as before. There is no clock in this case, and rejuvenation is attempted only when a node transits into the "degraded" state. In practice, this degraded state could be predicted in advance by means of analyses of observable system parameters as described above and in [11, 12]. Assume that prediction succeeds with probability c_2. If a prediction is successful, a token is deposited in place P_{detect}. If no other node is being rejuvenated at that time, the newly detected node can be rejuvenated. Once rejuvenation is completed, the token is put back in place P_{up}. Note that a node is allowed to fail even while waiting for rejuvenation. If the prediction is unsuccessful, a token is deposited in place $P_{detectfail}$ and then transits to the degraded state unchecked. Transitions $T_{nodefail1}$ and $T_{nodefail2}$ have the same rates.

Acknowledgments

The authors gratefully acknowledge the constructive comments of David Bradley of IBM and Katerina Goseva–Popstojanova of Duke University. This work was

supported by IBM as an enhancement project of the Center for Advanced Computing and Communication, Duke University.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Microsoft Corporation, Novell, Inc., The Santa Cruz Operation, Inc., or Sun Microsystems, Inc.

References

1. J. Gray and D. P. Siewiorek, "High-Availability Computer Systems," *IEEE Computer* **24,** 39–48 (September 1991).
2. M. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability—A Study of Field Failures in Operating Systems," *Proceedings of the 21st IEEE International Symposium on Fault-Tolerant Computing*, 1991, pp. 2–9.
3. Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications," *Proceedings of the 25th Symposium on Fault Tolerant Computer Systems*, Pasadena, CA, June 1995, pp. 381–390.
4. A. Avritzer and E. J. Weyuker, "Monitoring Smoothly Degrading Systems for Increased Dependability," *Empirical Software Eng. J.* **2,** No. 1, 59–77 (1997).
5. L. Bernstein, text of seminar delivered by Bernstein, University Learning Center, George Mason University, Fairfax, VA, January 29, 1996.
6. B. O. A. Grey, "Making SDI Software Reliable Through Fault-Tolerant Techniques," *Defense Electron.* **19,** 77–80, 85–86 (August 1987).
7. A. T. Tai, S. N. Chau, L. Alkalaj, and H. Hecht, "On-Board Preventive Maintenance: Analysis of Effectiveness and Optimal Duty Period," *Proceedings of the 3rd International Workshop on Object Oriented Real-Time Dependable Systems*, Newport Beach, CA, February 1997, pp. 40–47.
8. E. Marshall, "Fatal Error: How Patriot Overlooked a Scud," *Science*, p. 1347 (March 13, 1992).
9. G. Pfister, *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1998.
10. K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability and Programmability*, McGraw-Hill Book Co., Inc., New York, 1993.
11. S. Garg, A. van Moorsel, K. Vaidyanathan, and K. Trivedi, "A Methodology for Detection and Estimation of Software Aging," *Proceedings of the 9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, November 1998, pp. 282–292.
12. K. Vaidyanathan and K. S. Trivedi, "A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems," *Proceedings of the Tenth IEEE International Symposium on Software Reliability Engineering*, Boca Raton, FL, November 1999, pp. 84–93.
13. R. Chillarege, S. Biyani, and J. Rosenthal, "Measurement of Failure Rate in Widely Distributed Software," *Proceedings of the 25th IEEE International Symposium on Fault Tolerant Computing*, Pasadena, CA, July 1995, pp. 424–433.
14. R. K. Iyer and D. J. Rossetti, "Effect of System Workload on Operating System Reliability: A Study on IBM 3081," *IEEE Trans. Software Eng.* **SE-11,** No. 12, 1438–1448 (December 1985).
15. I. Lee, "Software Dependability in the Operational Phase," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana–Champaign, IL, 1995.
16. R. K. Iyer, L. T. Young, and P. V. K. Iyer, "Automatic Recognition of Intermittent Failures: An Experimental Study of Field Data," *IEEE Trans. Computers* **39,** No. 4, 525–537 (April 1990).
17. D. Tang and R. K. Iyer, "Dependability Measurement Modeling of a Multicomputer System," *IEEE Trans. Computers* **42,** No. 1 (January 1993).
18. A. Thakur and R. K. Iyer, "Analyze-NOW—An Environment for Collection and Analysis of Failures in a Network of Workstations," *Proceedings of the International Symposium on Software Reliability Engineering*, White Plains, NY, April 1996, pp. 14–23.
19. R. A. Maxion and F. E. Feather, "A Case Study of Ethernet Anomalies in a Distributed Computing Environment," *IEEE Trans. Reliability* **39,** No. 4 (1990).
20. S. Garg, A. Puliafito, and K. S. Trivedi, "Analysis of Software Rejuvenation Using Markov Regenerative Stochastic Petri Net," *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, Toulouse, France, October 1995, pp. 180–187.
21. S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi, "Time and Load Based Software Rejuvenation: Policy, Evaluation and Optimality," *Proceedings of the First Fault-Tolerant Symposium*, Madras, India, December 22–25, 1995.
22. S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi, "Minimizing Completion Time of a Program by Checkpointing and Rejuvenation," *Proceedings of the 1996 ACM SIGMETRICS Conference*, Philadelphia, PA, May 1996, pp. 252–261.
23. A. Pfening, S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, "Optimal Rejuvenation for Tolerating Soft Failures," *Perform. Eval.* **27 & 28,** 491–506 (October 1996).
24. S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, "Analysis of Preventive Maintenance in Transactions Based Software Systems," *IEEE Trans. Computers* **47,** No. 1, 96–107 (January 1998).
25. S. W. Hunter and W. E. Smith, "Availability Modeling and Analysis of a Two Node Cluster," *Proceedings of the 5th International Conference on Information Systems, Analysis and Synthesis*, Orlando, FL, October 1999.
26. V. B. Mendiratta, "Reliability Analysis of Clustered Computing Systems," *Proceedings of the Ninth IEEE International Symposium on Software Reliability Engineering*, Paderborn, Germany, November 1998, pp. 268–272.
27. M. R. Lyu and V. B. Mendiratta, "Software Fault Tolerance in a Clustered Architecture: Techniques and Reliability Modeling," *Proceedings of the 1999 IEEE Aerospace Conference*, Snowmass, CO, March 1999, pp. 141–150.
28. J. K. Muppala, G. Ciardo, and K. S. Trivedi, "Stochastic Reward Nets for Reliability Prediction," *Commun. RMS*, July 1994.
29. K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi, "Analysis of Software Rejuvenation in Cluster Systems," *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, ACM SIGMETRICS 2001/PERFORMANCE 2001*, Cambridge, MA, 2001, in press.
30. G. Ciardo, J. Muppala, and K. S. Trivedi, "SPNP: Stochastic Petri Net Package," *Proceedings of the Third International Workshop on Petri Nets and Performance Models*, Kyoto, Japan, 1989, pp. 142–151.
31. C. Hirel, B. Tuffin, and K. S. Trivedi, "SPNP: Stochastic Petri Nets. Version 6.0," *Computer Performance Evaluation: Modeling Tools and Techniques; 11th*

International Conference, TOOLS 2000, Schaumburg, IL, B. Haverkort, H. Bohenkamp, and C. Smidts, Eds., *Lecture Notes in Computer Science*, Vol. 1786, Springer Verlag, 2000, pp. 354–357.
32. C. Gage, IBM Secureway Network Dispatcher 2.1; *http://www-4.ibm.com/software/network/dispatcher/library/*.
33. H. Bozdogan, "Model Selection and Akaike's Information Criterion (AIC): The General Theory and Its Analytical Extensions," *Psychometrika* **53,** No. 3, 345–370 (1987).
34. N. R. Draper and H. Smith, *Applied Regression Analysis*, 2nd ed., John Wiley & Sons, Inc., New York, 1981.
35. K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.
36. C. Wang, D. Logothetis, K. S. Trivedi, and I. Viniotis, "Transient Behavior of ATM Networks Under Overloads," *Proceedings of IEEE INFOCOM*, Vol. 3, San Francisco, CA, March 24–28, 1996, pp. 978–985.

Received September 8, 2000; accepted for publication March 30, 2001

Vittorio Castelli *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (vittorio@us.ibm.com).* Dr. Castelli received a "Laurea" degree in electrical engineering from the Politecnico di Milano in 1988, with a thesis in bioengineering; he received an M.S. in electrical engineering in 1990, an M.S. in statistics in 1994, and a Ph.D. in electrical engineering in 1995 with a dissertation on information theory and statistical classification, all from Stanford University. In 1995 he joined the IBM Thomas J. Watson Research Center as a Postdoctoral Fellow, and became a Research Staff Member in 1996. From 1996 to 1998 he was co-investigator of NASA/CAN Project No. NCC5-101. Dr. Castelli is currently a member of the Systems Theory and Analysis group, where he works on memory compression, prediction, and performance analysis. His main research interests include information theory, statistics, and classification, and their applications to performance analysis and computer architecture. He is a member of Sigma Xi, the IEEE Information Technology Society, and the American Statistical Association. Dr. Castelli has published papers on computer-assisted instruction, statistical classification, data compression, image processing, multimedia databases, database mining, and multidimensional indexing structures.

Richard E. Harper *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (reharper@us.ibm.com).* Dr. Harper has been a Research Staff Member at the IBM Thomas J. Watson Research Center since 1998. Previously he was a Senior Technical Consultant at Stratus Computer in Marlboro, Massachusetts, from 1995 to 1998, and a Principal Member of the Technical Staff at the Charles Stark Draper Laboratory from 1987 to 1995. He received his Ph.D. in computer systems engineering from the Massachusetts Institute of Technology in 1987, his Master of Science degree in aeronautical engineering from Mississippi State University in 1978, and his Bachelor of Science degree in physics from Mississippi State University in 1977. His interests are in the design, analysis, and implementation of highly reliable high-performance computing systems.

Philip Heidelberger *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (philiph@us.ibm.com).* Dr. Heidelberger received a B.A. degree in mathematics from Oberlin College in 1974 and a Ph.D. degree in operations research from Stanford University in 1978. He has been a Research Staff Member at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York, since 1978. His research interests include modeling and analysis of computer performance, probabilistic aspects of discrete-event simulations, and parallel simulation. He has won Best Paper awards at the ACM SIGMETRICS and ACM PADS (Parallel and Distributed Simulation) Conferences, and he was twice awarded the INFORMS College on Simulation's Outstanding Publication Award. Dr. Heidelberger has recently served as Editor-in-Chief of the ACM's *Transactions on Modeling and Computer Simulation*. He is the General Chairman of the ACM SIGMETRICS/Performance 2001 Conference and has served as the Program Chairman of the 1989 Winter Simulation Conference and as the Program Co-Chairman of the ACM SIGMETRICS/Performance '92 Conference. He is a Fellow of the ACM and the IEEE.

Steven W. Hunter *IBM Server Group, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (hunters@us.ibm.com).* Dr. Hunter is a Senior Engineer in the xSeries Architecture and Technology Department of the IBM Server Group. In addition to software rejuvenation, he is currently involved with future high-density rack-mount systems, the Summit architecture, clustering, web-hosting and networking technology. Prior to this, he was in the IBM Networking Hardware Division, where he worked on numerous networking products and high-speed networking technology. Dr. Hunter received the B.S. degree in electrical engineering from Auburn University, the M.S. degree in electrical and computer engineering from North Carolina State University, and the Ph.D. degree in electrical and computer engineering from Duke University. His research interests include systems and networking architecture, design, analysis, and implementation. He is also an adjunct professor at North Carolina State University, where he currently teaches a course on high-speed networking.

Kishor S. Trivedi *Center for Advanced Computing and Communication, Department of Electrical and Computer Engineering, Duke University, Durham, North Carolina 27708 (kst@ee.duke.edu).* Dr. Trivedi received the B.Tech. degree from the Indian Institute of Technology (Bombay), and M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana–Champaign. He holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University, and has a joint appointment in the Department of Computer Science at Duke. He has been on the Duke faculty since 1975. Dr. Trivedi is the Duke Site Director of an NSF industry–university cooperative research center between North Carolina State University and Duke University for carrying out applied research in computing and communications. He has served as a Principal Investigator on various projects funded by AFOSR, ARO, Burroughs, DARPA, the Draper Laboratory, IBM, DEC, Alcatel, Telcordia, Motorola, NASA, NIH, ONR, NSWC, Boeing, Union Switch and Signals, NSF, and SPC, and as a consultant to industry and research laboratories. Dr. Trivedi was an Editor of the *IEEE Transactions on Computers* from 1983 to 1987. He is a co-designer of the HARP, SAVE, SHARPE, SPNP, and SREPT modeling packages, which have been widely circulated. He is the author of the book *Probability and Statistics with Reliability, Queuing and Computer Science Applications* (Prentice Hall) and has recently published two books entitled *Performance and Reliability Analysis of Computer Systems* (Kluwer Academic Publishers) and *Queueing Networks and Markov Chains* (John Wiley). Dr. Trivedi's research interests are in reliability and performance assessment of computer and communication systems. He has published more than 250 articles and lectured extensively on these topics, and has supervised 32 Ph.D. dissertations. He is a Fellow of the Institute of Electrical and Electronics Engineers and a Golden Core Member of the IEEE Computer Society.

Kalyanaraman Vaidyanathan *Center for Advanced Computing and Communication, Department of Electrical and Computer Engineering, Duke University, Durham, North Carolina 27708 (kv@ee.duke.edu).* Mr. Vaidyanathan received his bachelor's degree in computer science from the University of Madras, India, in 1996 and the master's degree in electrical and computer engineering from Duke University in 1999. He is currently a Ph.D. student at Duke and received the IBM Graduate Fellowship Award in 2000. His research interests include fault-tolerant systems, software reliability and performance, and dependability evaluation of computer systems.

William P. Zeggert *IBM Server Group, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (wzeggert@us.ibm.com).* Mr. Zeggert received his B.S. degree in computer science from Appalachian State University in 1993. His undergraduate research included a statistical library which assisted in proving various statistical characteristics, specifically in an M-Erlang distribution. Mr. Zeggert has experience as a software development consultant in the financial, telecommunication, and network administration industry, developing cross-platform heterogeneous systems for banking and insurance transaction processing, European cellular phone networks and networking/SNMP event/problem tracking frameworks as developmental backbone for large-scale software systems. Mr. Zeggert has also worked as a civil engineering research scientist providing computational expertise within the research engineering community. He researched and developed optimization algorithms and 3D/OpenGL modeling in munitions effects assessment and worked on the development of a Linux parallel computational system to streamline statistical processing of wind dynamics. Mr. Zeggert is now an IBM software engineer providing leadership and development expertise for Intel-hardware-specific system administration utilities including three-tier Java/Swing-based software providing flexible and portable administration tools such as Fuel Gauge, Cluster Systems Management, System Availability, and Software Rejuvenation. He provided the presentation mechanism for software rejuvenation and assisted in the proving and viability of rejuvenation algorithms in the Microsoft OS environment. Mr. Zeggert is currently leading the software quality initiative in the IBM eSeries. His role includes ensuring that the eSeries software, including but not limited to software rejuvenation, is developed so that the customer is provided with the highest quality of software at the lowest possible cost. This cost is measured in development, test, and support dollars.