Software Rejuvenation: Analysis, Module and Applications

Yennun Huang and Chandra Kintala
AT&T Bell Laboratories
Murray Hill, NJ 07974

Nick Kolettis and N. Dudley Fulton
AT&T Bell Laboratories
Middletown, NJ 07748

Abstract

Software rejuvenation is the concept of gracefully terminating an application and immediately restarting it at a clean internal state. In a client-server type of application where the server is intended to run perpetually for providing a service to its clients, rejuvenating the server process periodically during the most idle time of the server increases the availability of that service. In a long-running computation-intensive application, rejuvenating the application periodically and restarting it at a previous checkpoint increases the likelihood of successfully completing the application execution. We present a model for analyzing software rejuvenation in such continuously-running applications and express downtime and costs due to downtime during rejuvneation in terms of the parameters in that model. Threshold conditions for rejuvenation to be beneficial are also derived.

We implemented a reusable module to perform software rejuvenation. That module can be embedded in any existing application on a UNIX platform with minimal effort. Experiences with software rejuvenation in a billing data collection subsystem of a telecommunications operations system and other continuously-running systems and scientific applications in AT&T are described.

†Proceedings of the 25th Intl. Symposium on Fault-Tolerant Computing (FTCS-25), Pasadena, CA, pp. 381-390, June 27-30, 1995.

1 Introduction

The server processes in most client-server software systems are intended to run continuously forever except perhaps during software upgrades. This requirement is however very difficult, if not impossible, to design and guarantee. It is particularly difficult in applications of nontrivial complexity, whether those applications are built from ground up or through a combination of reuse, vendor software components, and new development.

Elusive bugs of *non-deterministic* character, or of root origin difficult to characterize, could stay inactive in such applications for a long time before they surface; Gray[3] calls such bugs *Heisenbugs*[1] (as opposed to *Bohr bugs* for faults that predictably lead to failures). We are not speaking here of faults which should have been caught and eliminated through genuinely diligent implementation and testing, but rather of those faults that have escaped all possible analysis and testing and are likely to manifest in protracted executions of applications, eventually interfering with their longevity. Bernstein[2] advocates a study of such faults in telecommunication systems; Huang[5] provides a more detailed characterization and description of those faults.

When such software bugs (faults) are encountered during executions, they could lead to *transient* failures with unpredictable and costly after effects. For example, they might corrupt a database far beyond repair without leaving a trail; they might cause memory leakage or bloating (e.g., from repeated allocation/deallocation cycles), either of which will eventually crash the process; or they might induce slow choking of other operating system resources, eventually paralyzing an entire application.

The strategies that are typically employed to handle such transient failures are *reactive* in nature, i.e., they consist of actions *after a failure*. Until recently, most such solutions were implemented either manually or through "clever" programming, idiosyncratic to each application. Recently, platform and application independent modules, `watchd`, `libft` and `nDFS/REPL`,

[1] Program debugging community uses the term *Heisenbug* for bugs that go away when the debugging or tracing flags are turned on during program compilation. Software fault tolerance community uses that term to characterize the bugs that we are discussing. These two classes of bugs may have some overlap but are certainly different.

have been used in several applications to recover from transient failures *after* they are detected[4]. In all such reactive solutions, if an application process fails, some recovery mechanism in place attempts to respawn it. Some mechanisms, such as `watchd` and `libft`, rollback the process to its last checkpoint, if such a rollback makes sense. Newer recovery mechanisms include reordering the logged messages and replaying them after rollback[5, 10].

Such reactive fault tolerance mechanisms (restart, recovery, rollback, reorder and replay) succeed quite often and should be implemented wherever possible to ensure high availability and data integrity of long-running applications. They also help to recover from failures which may occur outside the application domain - for example, when a telecommunication network communications facility is cut, the application must go into recovery. However, those reactive strategies do not always provide a uniformly satisfying mechanism to ensure longevity with a high degree of confidence - the restoration may fail even after all possible attempts to reorder and replay, thereby shortening longevity even further!

We propose a complementary approach to handle transient software failures. It is a *preventive* and *proactive* (as opposed to being a reactive) solution and we call it *software rejuvenation*. This technique is particularly useful for the faults due to resource leaking or deadlock. We must emphasize that this preventive solution **does not** replace the other failure recovery mechanisms but complements them by reducing, but not completely eliminating, the possibilities for such failures to occur. When those failures do occur in spite of rejuvenation, the other reactive mechanisms must be employed to recover the application.

2 Software Rejuvenation

When software applications execute continuously for long periods of time (scientific and analytical applications run for days or weeks, servers in client-server systems are expected to run forever), the processes corresponding to the software in execution age or slowly degrade with respect to effective use of their system resources. The causes of *process aging* are memory leaking, unreleased file locks, file descriptor leaking, data corruption in the operating environment of system resources, etc. Process aging will affect the performance of the application and eventually cause the application to fail.

Of course, if the application software was developed in a perfect development environment and was proved to be correct in all possible execution scenarios then the executing processes corresponding to that software will not age. But, practical software systems are rarely that perfect. So, their processes do age during their execution. We should also point out that process aging is different from software aging as defined by Parnas[8]. Software aging is related to the application program source becoming obsolete due to changing requirements and maintenance over the years. Process aging is related to the application processes getting degraded over days and weeks of execution.

Resource leaking and other problems causing the processes to age are due to bugs in the application program that was developed, in the libraries that the application is using or in the application execution environment (e.g. operating system). Fixing all those program bugs is not always possible because, for example, the bug could be in the library code or reused code for which the application developer may not have access to the source. For complex software applications, sometimes it is also very difficult to identify the source of those bugs.

Process aging does not make an application fail immediately after the application is started. It takes time for an application to age and then eventually crash. Therefore, there is an interval after an application is initialized such that the application is healthy and the probability of a failure is almost zero. We call this interval the application's *base longevity interval*. For example, if a process (application) has a memory leaking problem, the process will not fail when its size is less than, say, 10M bytes. Therefore, the base longevity interval for the process is the mean time for its size to grow from an initial state to a state with its size over 10M bytes. Similarly, a telecommunication system which allocates voice links and has a software bug in its boundary condition may never fail when it still has more than 10 links available. Therefore, the base longevity interval of the system is the mean time for the system to allocate its first voice link to the time it has only 10 links left. The base longevity interval can be measured by putting a resource monitor or a state monitor into an application. We will not elaborate here on the methods that might be used to determine that interval, although it is expected that they would relate to the process by which the application was developed, tested and delivered. If the application can indeed run flawlessly for its base longevity interval every time it starts, then it is reasonable to expect that a *preemptive* scheduled restart of the application at a frequency determined by its base longevity interval would make that application available *much*

longer!

Software rejuvenation is this concept of periodically and preemptively restarting an application at a clean internal state after every rejuvenation interval. The rejuvenation interval is based on experience with the application and its base longevity interval. Restarting the application involves queuing the incoming messages temporarily, cleaning up the in-memory data structures, respawning the processes at the initial state or at a previously checkpointed state, logging administrative records, etc. Thus,

> **software rejuvenation is periodic preemptive rollback of continuously running applications to prevent failures in the future.**

The server in a client-server application will of course be unavailable during rejuvenation. So rejuvenation may sometimes increase the downtime of an application. But those are planned and scheduled downtimes. If care is taken to schedule rejuvenations during the most idle times of an application, then the *cost* due to those downtimes is expected to be low. Downtime costs are the costs incurred due to the unavailability of the service during downtime of an application. In a telecommunication application, they include the costs due to the loss of business during downtime and the time and material costs for recovery. Such costs often depend on when the service goes down. They are likely to be high if the downtime is unscheduled as it happens during a failure. They are expected to be much lower during scheduled downtimes as during rejuvenations.

In Section 3, we present an analytical model to capture the above concepts, notion and intuition behind software rejuvenation and derive the expected costs due to downtime of an application with and without rejuvenation. Based on those estimations, we compute the threshold conditions under which software rejuvenation becomes cost-effective. It turns out that, if those threshold conditions hold, rejuvenation interval does not influence the decision to rejuvenate but effects only the reduction in downtime costs due to rejuvenation. We provide three examples in Secton 3.3 to illustrate how a decision to rejuvenate is to be made.

Rejuvenation, per se, is perhaps not an innovative concept in the computing domain. However, tales related to rejuvenations are usually in the context of hardware failures. Folklore in software is abundant and some of it includes the basic notions described here[9]. But, the authors could not find a reference to this particular term "software rejuvenation" or concept in the published literature on software fault tolerance and reliability[2]. Software rejuvenation, if it is ever done, is usually programmed into an application individually. Here, we extract out all the essential and generic tasks needed for embedding software rejuvenation in an application and encapsulate them in a reusable module. We describe that reusable module in Section 4 and discuss specific experiences with rejuvenation, a client-server application and a long-running scientific application, in that section.

3 Model and Analysis

In this section, we are interested in determining how software rejuvenation can reduce an application's downtime and the cost due to downtime. Intuitively, an application which never fails should not be rejuvenated. On the other hand, if an application always fails on the eighth day after it starts and the rejuvenation cost is very small, we should rejuvenate the application every 7 days so that no failure will occur. In general, the decision about rejuvenation depends on the rejuvenation cost, the application failure rate and the downtime cost. We present a model and analyze it for rejuvenation in terms of those parameters and derive the threshold conditions. In the following, we use A to denote an application without rejuvenation and A^r to denote the application with rejuvenation.

3.1 Downtime and Cost due to Downtime

The performance of an application, i.e. the availability of its service, is of course impaired during rejuvenation. So, the cost of downtime during rejuvenation must be taken into account to make a decision on rejuvenation. Since rejuvenation involves scheduled downtime, the cost of that downtime is expected to be much lower than the cost of an unscheduled downtime caused by a failure. To compute downtime costs, we first look at the probabilistic state transition diagram of an application A without rejuvenation as shown in Figure 1. When it starts, the application stays in a highly robust state S_0 for a period corresponding to

[2] We came across a recent reference to the term *software rejuvenation* in the paper[6]; however the term is used in the context of software reengineering and reuse, not in the sense we use it here. The paper[1] discusses installing bug fixes, not preemptive clean-up of an executing process as we discuss here. Musa[7] makes a passing remark, in Page 79 of his book, on periodic reinitialization of data in the context of software reliability but does not study it further.

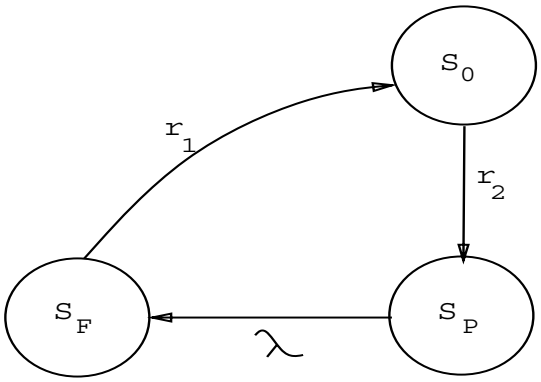

Figure 1: Probabilistic state transition model for A without rejuvenation

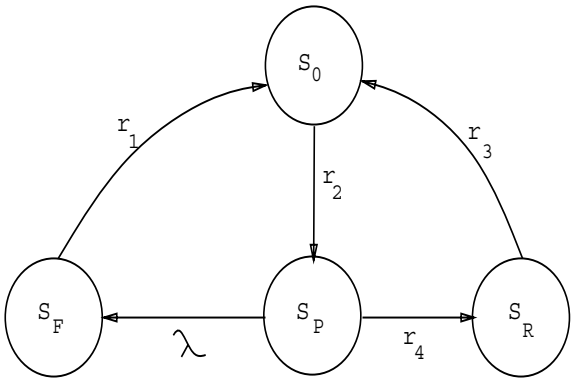

Figure 2: Probabilistic state transition model for A with rejuvenation

its base longevity interval; then it goes into a failure probable state S_P. This is because, in our experience, a well-tested software application stays "healthy" for a while before it reaches a state where failures are probable; it often takes a while for a program to reach its boundary conditions or leak out some of its resources.

Thus, failure is a 2-step behavior as shown in Figure 1 where an application A goes from state S_0 to state S_P at a probabilistic mean rate of r_2 and from state S_P to state S_F at a probabilistic mean rate of λ. Observe that $\frac{1}{r_2}$ corresponds to the base longevity interval for the application as discussed in the previous section. When A is repaired after a failure, it goes from state S_F to state S_0. We assume that the repair time for the application is also exponentially distributed with a rate r_1. We also assume that the probability of going from state S_0 to failure state S_F is negligible compared to the other probabilities.

Under these assumptions and solving the equations $p_0 + p_p + p_f = 1$, $p_p \cdot \lambda = p_0 \cdot r_2$, and $p_f \cdot r_1 = p_p \cdot \lambda$ (where p_0, p_p, p_f denote the probabilities of the system being in states S_0, S_P, and S_F, respectively) we derive p_f, the steady-state unavailability factor for the system, to be equal to $\frac{1}{1+\frac{r_1}{\lambda}+\frac{r_1}{r_2}}$. So, the expected total downtime of A in an interval of L time units is:

$$DownTime_A(L) = \frac{1}{1+\frac{r_1}{\lambda}+\frac{r_1}{r_2}} \times L \qquad (1)$$

When an application is down, no service is provided and no revenue is received. Therefore, there is a business cost due to service being unavailable during downtime. The cost varies from service to service. If c_f is the average cost per unit of A's unscheduled downtime, then the expected cost of downtime in an interval of L time units is:

$$Cost_A(L) = \frac{1}{1+\frac{r_1}{\lambda}+\frac{r_1}{r_2}} \times L \times c_f \qquad (2)$$

Now consider the probabilistic state transition diagram of application A with rejuvenation as shown in Figure 2 where state S_R is the rejuvenation state and the other states are as before. We assume that the rejuvenation rate r_4 and the repair rate r_3 after a rejuvenation event are also exponentially distributed. If the application is rejuvenated after every t units then r_4 will be equal to $1/t$. Solving the probability equalities generated from this model for A with rejuvenation, we get the following expressions for state probabilities:

$$\begin{aligned} p_p &= \frac{1}{1+\frac{\lambda}{r_1}+\frac{r_4}{r_3}+\frac{\lambda+r_4}{r_2}} \\ p_f &= \frac{\lambda}{r_1} \cdot p_p \\ p_r &= \frac{r_4}{r_3} \cdot p_p \\ p_0 &= \frac{\lambda + r_4}{r_2} \cdot p_p \end{aligned}$$

The expected total downtime of the application A with rejuvenation in an interval of L time units is:

$$DownTime_{A^r}(L) = (p_f + p_r) \times L \qquad (3)$$

If c_f is the average per unit cost of unscheduled downtime as before and c_r is the average per unit cost of downtime during rejuvenation, then the total expected downtime cost in an interval of L time units is:

$$
\begin{aligned}
Cost_{A^r}(L) &= (p_f \times c_f + p_r \times c_r) \times L \\
&= \frac{L}{1 + \frac{\lambda}{r_1} + \frac{r_4}{r_3} + \frac{\lambda + r_4}{r_2}} \\
&\quad \times (\frac{\lambda}{r_1} \times c_f + \frac{r_4}{r_3} \times c_r) \qquad (4)
\end{aligned}
$$

We can verify that if rejuvenation is not performed then $r_4 = 0$, $DownTime_{A^r}(L) = DownTime_A(L)$ in Equations 1 and 3 and $Cost_{A^r}(L) = Cost_A(L)$ in Equations 2 and 4. If rejuvenation is performed during the most idle time of the application, then $r_3 > r_1$ and $c_f >> c_r$ and hence the total expected downtime cost in the application with rejuvenation, as computed in Equation 4, would be lower than that without rejuvenation as computed in Equation 2. Such thresholds for rejuvenation are discussed in the next section.

3.2 Rejuvenation Thresholds

Intuitively, if r_3 is very large, i.e. fast recovery from rejuvenation state, rejuvenation should reduce the total downtime of an application since the application can stay in state S_0 (healthy state) more often and the chance of a failure becomes smaller. On the other hand, if r_3 is very small, i.e. slow recovery from rejuvenation state, rejuvenation of an application increases the time that the application stays in state S_R, and hence, the total system downtime increases. Similarly, if the application failure rate λ is very large, rejuvenation should be performed frequently so that the application stays in state S_0 longer, and hence, the chance of a failure becomes smaller. However, if the application fails very rarely, rejuvenation increases the total application downtime. Similar threshold effects also exist for the base longevity interval ($\frac{1}{r_2}$) and the application repair rate after a failure (r_1). We study such thresholds below.

Let us analyze how the downtime and cost of downtime change when the rejuvenation rate, r_4, changes. Substituting the values for p_f and p_r, Equation 3 can be written as

$$
DownTime_{A^r}(L) = L \times \frac{\frac{\lambda}{r_1} + \frac{r_4}{r_3}}{1 + \frac{\lambda}{r_1} + \frac{\lambda}{r_2} + \frac{r_4}{r_2} + \frac{r_4}{r_3}} \qquad (5)
$$

To examine the behavior of downtime when r_4 changes, we need to differentiate the above equation with respect to r_4. Observe that the numerator and the denominator in Equation 5 are linear functions of r_4. So, the differentiation of the downtime function yields:

$$
\begin{aligned}
\frac{d}{dr_4} DownTime_{A^r}(L) &= L \times \frac{\lambda}{r_1 r_2 r_3} \\
&\quad \times \frac{1}{(1 + \frac{\lambda}{r_1} + \frac{\lambda}{r_2} + \frac{r_4}{r_2} + \frac{r_4}{r_3})^2} \\
&\quad \times [r_1(1 + \frac{r_2}{\lambda}) - r_3] \qquad (6)
\end{aligned}
$$

It is interesting to note that the denominator in the above derivative is always positive and that the sign of the numerator is determined by the expression $[r_1(1+\frac{r_2}{\lambda}) - r_3]$ which is independent of r_4. This means that, when r_4 changes, the downtime increases or decreases depending entirely on the values of λ, r_1, r_2 and r_3. When r_3 is greater than $r_1(1+\frac{r_2}{\lambda})$, the derivative is negative implying that the downtime always decreases when the value of r_4 increases. This implies that we should do rejuvenation as soon as the application goes into state S_P. This conforms to our intuition that if the cost of rejuvenation is small and the failure rate is large, we should do rejuvenation as soon as the application goes into a failure probable state. For example, if an application could potentially fail when its process size is over 40M bytes, the above condition implies that we should do rejuvenation as soon as its size reaches 40M bytes. Similarly when r_3 is less than $r_1[1 + \frac{r_2}{\lambda}]$, i.e. slow recovery from rejuvenation state, the derivative is positive and the downtime always increases when the rejuvenation interval r_4 increases.

Now let us examine Equation 4 to determine the behavior of the expected downtime cost when the rejuvenation rate r_4 changes. For simplicity, we assume c_r is independent of r_4 and differentiate the $Cost_{A^r}$ function in Equation 4 with respect to r_4 to obtain [3]

$$
\begin{aligned}
\frac{d}{dr_4} Cost_{A^r}(L) &= L \times \frac{1}{r_1 r_2 r_3} \times \frac{1}{(r_1 r_2 + \lambda r_1 + \lambda r_2)} \\
&\quad \times \frac{1}{(1 + \frac{\lambda}{r_1} + \frac{\lambda}{r_2} + \frac{r_4}{r_2} + \frac{r_4}{r_3})^2} \\
&\quad \times [c_r - c_f \frac{\lambda(r_2 + r_3)}{\lambda(r_1 + r_2) + r_1 r_2}] \qquad (7)
\end{aligned}
$$

Again, the denominator in the above derivative is always positive and the sign of the numerator is determined by the expression $[c_r - c_f \frac{\lambda(r_2+r_3)}{\lambda(r_1+r_2)+r_1 r_2}]$ which is independent of r_4. This means that, when

[3] In practice, c_r is independent of r_4 only for a small range of values of r_4. If r_4 is large, increasing r_4 will also increase c_r because the rejuvenation will occur during a busy period.

r_4 changes, the total expected downtime cost increases or decreases depending entirely on the values of $c_r, c_f, \lambda, r_1, r_2$ and r_3.

This brings us to a very interesting observation. From the application's downtime cost viewpoint, the decision as to whether an application should be rejuvenated or not does not depend on the rate r_4 with which that application is rejuvenated but on the other parameters in the model. For example, the rejuvenation and failure costs c_r and c_f of an application might be such that the condition $c_r < c_f[\frac{\lambda(r_2+r_3)}{\lambda(r_1+r_2)+r_1r_2}]$ is satisfied. Then the slope of the cost function with respect to r_4 is negative implying that when r_4 is increased, the total expected downtime cost decreases. This means that the application benefits from rejuvenation. In this case, given that the total cost continues to decrease when r_4 is increased, it is always better to perform rejuvenation as soon as the system enters its failure probable state S_P if the condition $c_r < c_f[\frac{\lambda(r_2+r_3)}{\lambda(r_1+r_2)+r_1r_2}]$ is satisfied. Similarly, consider an application with a rejuvenation cost c_r greater than $c_f[\frac{\lambda(r_2+r_3)}{\lambda(r_1+r_2)+r_1r_2}]$. If rejuvenation is performed on this application, the total cost increases when the rate of rejuvenation r_4 is increased. This implies that this application will not benefit from any rejuvenation at all.

The above discussion illustrates that there is a threshold effect. When $r_4 = 0$, there is no rejuvenation and the downtime and cost values can be computed as shown earlier. When r_4 is increased, the downtime increases or decreases depending entirely on whether the condition $r_3 < r_1(1 + \frac{r_2}{\lambda})$ is satisfied or not. Similarly, when r_4 is increased and c_r is independent of r_4, the cost due to downtime increases or decreases depending entirely on whether the condition $c_r > c_f[\frac{\lambda(r_2+r_3)}{\lambda(r_1+r_2)+r_1r_2}]$ is satisfied or not. Both conditions are independent of r_4. The downtime and the cost functions continue to increase or decrease as long as those conditions hold. For the best results, we should perform rejuvenation as soon as an application enters a failure probable state, i.e. make $r_4 = \infty$, (assuming that the cost functions do not change) or do not perform rejuvenation at all, i.e. make $r_4 = 0$.

3.3 Illustrative Examples

Let us now look at some examples to illustrate benefits of rejuvenation quantitatively.

3.3.1 Example A

Consider an application A with the following "failure profile".

- MTBF, the mean time between two consecutive failures, is 12 months; so, $\lambda = 1/(12 \times 30 \times 24)$.
- It takes 30 minutes to recover from an unexpected failure; so $r_1 = 2$.
- The base longevity interval, which is the time for the application to go from state S_0 to state S_P, is 7 days. Then, $r_2 = \frac{1}{7\times24}$.
- If rejuvenation is performed, the mean repair time after rejuvenation is 20 minutes. That is, $r_3 = 3$.
- The average cost of unscheduled downtime due to a failure, c_f, is \$1,000 per hour.
- The average cost of scheduled downtime during rejuvenation, c_r, is \$40 per hour.

Let us first consider the hours of downtime. The expression $r_1[1 + \frac{r_2}{\lambda}]$ in Equation 6 can be computed to be 104.86. Since r_3 is 3 and is much less than the previous value, the downtime will increase when r_4 is increased. Similarly, when the threshold for the cost function is computed, we find that $c_f \frac{\lambda(r_2+r_3)}{\lambda(r_1+r_2)+r_1r_2}$ evaluates to \$28.67. The cost of rejuvenation c_r is \$40 per hour and this means that, when r_4 is increased, the cost due to downtime will increase. Thus both the downtime and cost due to downtime increase when r_4 is increased above zero and so rejuvenation is not beneficial in this application. Note that r_4 is the rate of rejuvenation after the application goes into the failure probable state. Therefore, if an application is rejuvenated every two weeks, r_4 is $\frac{1}{(14-7)*24}$. The table in Figure 3 illustrates how those values increase when the rejuvenation rate is increased.

3.3.2 Example B

Consider another application B with the following "failure profile".

- MTBF, the mean time between two consecutive failures, is 3 months; so $\lambda = 1/(3 \times 30 \times 24)$.
- It takes 30 minutes to recover from an unexpected failure; so $r_1 = 2$.
- It goes from the initial robust state S_0 to its failure probable state S_P in 3 days, i.e. its base longevity interval is 3 days; then $r_2 = \frac{1}{3\times24}$.

$\lambda = 1/(12 \times 30 \times 24)$, $r_1 = 2$, $r_2 = \frac{1}{7 \times 24}$, $r_3 = 3$, $c_f = \$1000$, $c_r = \$40$. For $L = 12 \times 30 \times 24$ hours and a rejuvenation frequency of			
	no rejuvenation ($r_4 = 0$)	once every 3 weeks ($r_4 = \frac{1}{2 \times 7 \times 24}$)	once every two weeks ($r_4 = \frac{1}{1 \times 7 \times 24}$)
Hours of Downtime	0.490	5.965	8.727
$Cost of Downtime	490	554	586

Figure 3: Failure Profile of Application A

$\lambda = 1/(3 \times 30 \times 24)$, $r_1 = 2$, $r_2 = \frac{1}{3 \times 24}$, $r_3 = 6$, $c_f = \$5000$, $c_r = \$5$. For $L = 12 \times 30 \times 24$ hours and a rejuvenation frequency of			
	no rejuvenation ($r_4 = 0$)	once every 2 weeks ($r_4 = \frac{1}{11 \times 24}$)	once a week ($r_4 = \frac{1}{4 \times 24}$)
Hours of Downtime	1.94	5.70	9.52
$Cost of Downtime	9,675.25	7672.43	5643.31

Figure 4: Failure Profile of Application B

- If a rejuvenation is performed, the mean repair time after rejuvenation is 10 minutes; so $r_3 = 6$.
- The average cost of unscheduled downtime due to a failure, c_f, is \$5,000 per hour.
- The average cost of scheduled downtime during rejuvenation, c_r, is \$5 per hour.

In this case, $r_1[1+\frac{r_2}{\lambda}]$ evaluates to 62 and r_3 is equal to 6 which is less than 62. This means, the downtime will increase when r_4 is increased. When we consider the threshold for the cost function, we find that $c_f \frac{\lambda(r_2+r_3)}{\lambda(r_1+r_2)+r_1 r_2}$ evaluates to \$484.67. The cost of rejuvenation c_r is \$5 per hour and this means that when r_4 is increased, the average total cost due to downtime will decrease. Thus, in this example, although the downtime increases when r_4 is increased, the cost due to downtime decreases when r_4 is increased. So, if cost is the only factor to decide on rejuvenation, it is beneficial to perform rejuvenation as frequently as possible, as long as the above threshold condition is not violated. The table in Figure 4 illustrates this behavior in this application.

3.3.3 Example C

Let us consider a third application C with the same parameters as in Example B with the following modifications: the base longevity interval is 10 days; that is $r_2 = \frac{1}{10 \times 24}$; and it now takes 2 hours to recover from an unexpected failure; that is, $r_1 = 0.5$. The expression $r_1[1 + \frac{r_2}{\lambda}]$ evaluates to 5. The value of r_3 is larger and hence the downtime will now decrease when r_4 is increased. Similarly, the expression $c_f[\frac{\lambda(r_2+r_3)}{\lambda(r_1+r_2)+r_1 r_2}]$ evaluates to \$2858. The cost of rejuvenation c_r is \$5 per hour and hence, when r_4 is increased, the cost due to downtime will decrease. Thus rejuvenation decreases both the downtime and the cost due to downtime in this example and hence it is beneficial. These benefits for this application are illustrated in the table in Figure 5.

4 Module and Applications

Software rejuvenation can be implemented in a simple application easily using UNIX[4] cron command after the rejuvenation interval is determined using the analysis techniques described in the previous section. For more complex continuously-running applications however, the internal data structures may need to be checkpointed before rejuvenation and restored after rejuvenation and a graceful shutdown and restart might be necessary to do rejuvenation. For such applications, we have enhanced the software fault tolerance component, *watchd*, to perform software rejuvenation. In this section, we describe that reusable module and describe a few applications that perform rejuvenation.

[4] UNIX is a registered trademark of X/Open Co.

$\lambda = 1/(3\times30\times24)$, $r_1 = 0.5$, $r_2 = \frac{1}{10\times24}$, $r_3 = 6$, $c_f = \$5000$, $c_r = \$5$. For $L = 12\times30\times24$ hours and a rejuvenation frequency of			
	no rejuvenation ($r_4 = 0$)	once every month ($r_4 = \frac{1}{(30-10)\times24}$)	once every two weeks ($r_4 = \frac{1}{(14-10)\times24}$)
Hours of Downtime	7.19	6.83	6.36
$Cost of Downtime	3.6K	2.48K	1.11K

Figure 5: Failure Profile of Application C

`Watchd` is a reusable watchdog daemon process to facilitate the detection of and recovery from application process failures[4]. It runs on a single machine or on a network of machines and continually watches the life of a local application process by periodically sending a null signal to the process and checking the return value to detect whether that process is alive or dead. It also detects whether that process is hung or not by either sending null messages or monitoring for heartbeats from the application. `Watchd` and other reusable fault tolerance components, `libft` and `nDFS/REPL`, have been used to enhance the availability of several applications in AT&T. They have been ported to several hardware and OS platforms including SunOS, SGI IRIX, HP-UX, Motorola, Lynx, NCR SVR4 etc.

4.1 A Reusable Module

`Watchd` has been modified to perform software rejuvenation. To rejuvenate a process using `watchd`, a user invokes the `addrejuv` command to stop and restart a process at a given time. The syntax of `addrejuv` command is:

```
addrejuv proc_name <cmd|sig>[:elapsed time]
          <sig>[:elapsed time] <time|event>
```

`Addrejuv` takes four arguments - process name, a command or signal number, another signal number and the time at which the rejuvenation will occur. Each command or signal number can be followed by an elapsed time to specify the duration between the two signals sent; the default value for that duration is 15 seconds. `Watchd` looks at its internal process table and finds the `pid` number of the specified process. Then, using that `pid` information, `watchd` creates a shell script and registers the starting time or the event for execution of the script with the `cron` daemon. When the time or event comes, `cron` daemon executes the shell script to rejuvenate the process. The rejuvenation shell script takes three steps to stop the process: first, a signal is sent to the process if the second argument in the `addrejuv` command is a signal number or a command is executed if that argument is a command name. Then, fifteen seconds later, the signal specified in the third argument of the `addrejuv` command is sent to the process; Finally, fifteen seconds later, a SIGKILL signal is sent to the process to make sure that the process is really terminated. The fifteen seconds interval between the two signals allows the process to clean up its state before being terminated; the default value of fifteen seconds can be changed by the application. Once the process is terminated and rejuvenation is complete (the last signal is sent to the process), `watchd` takes a recovery action to respawn the process in the same manner as it does when it detects a failure[4].

The above module provides a mechanism to rejuvenate a process by terminating that process and restarting it from a clean initial state. This is usually satisfactory for client-server applications where the critical state of the server is in permanent storage and updated after each transaction. In long-running applications, however, part of the application state is in volatile memory and hence the application needs to be checkpointed before rejuvenation. Programmer-assisted checkpointing[4] or transparent checkpointing[11] needs to be used to checkpoint the application state periodically before rejuvenating the application; see reference[11] for further details on how to do transparent checkpointing in such applications using a reusable library module.

4.2 A Telecommunications Billing Data Application

Rejuvenation is implemented in the BILLDATS II©Collector, a billing data collection system deployed throughout the AT&T long-distance network, and in several of the RBOCs (Regional Bell Operating Companies) and ITCOs (Independent Telephone Companies). Based on pre-deployment laboratory testing with longevity runs approximating 2

weeks, the rejuvenation interval in that system is conservatively set to 1 week for field installations. After several years of field operation of the BILLDATS II Collector *with rejuvenation*, not a single incidence of the type of failures described in Section 1 affecting longevity was encountered to date.

The concepts and the programming tasks that were needed to do rejuvenation in the BILLDATS II Collector were extracted, slightly modified and embedded in the reusable module `watchd` as described in the previous section. This will help other applications to perform rejuvenation with much less effort.

4.3 A Long-running Scientific Application

S is a scientific speech synthesis system that is expected to run continually for days to process hundreds of sentences. Processing each sentence involves complex data structure manipulations which in turn need data stored in files containing the analysis information from previous sentences. Due to a hideous bug in a vendor-supplied component in that application, the program used to run out of file descriptors and crash after processing about hundred sentences. Restarting the system after a failure would require reprocessing all the previous sentences. S has now been modified to perform rejuvenation periodically after processing every 15 sentences. Rejuvenation in this application involves suspending the process, taking a transparent checkpoint, performing garbage collection and resuming the process execution at the checkpointed state; see reference[11] for further details. With rejuvenation, the system is now able to run for days without a failure. The fault in the program still remains but, until it is fixed and confidence in the system's reliability is very high, it is more productive to use rejuvenation in that system to prevent failures due to that fault.

4.4 Slowly Degrading Systems

Some call-processing type of telecommunication systems exhibit *slow degradation* in performance and software resource utilization. Domain-based reliability measures have been applied to consider the effects of degradation on the expected reliability of those systems[12]. Work is underway to embed and monitor those measures during execution of such a system. Thresholds on the degradation measure can then be designed to trigger rejuvenation in the executing system.

5 Concluding Remarks

Observe that the rejuvenation frequency in an application can be of three possible types. It is based on time in the application discussed in Section 4.2, program logic in the application in Section 4.3 and a performance measure in the application in Section 4.4 above. The model and analysis we presented in Section 3 is helpful to determine whether rejuvenation will be useful or not.

Rejuvenation does not have to be applied uniformly to every process in an application. One can decide which of the application component processes are vulnerable to longevity flaws and choose them for rejuvenation. Service daemon processes are good candidates since they are typically expected to run forever. With proper analysis, selection and tuning of those parameters, as described in Section 3, it should be possible to obtain high availability and reduced cost due to downtime in many applications.

Obviously, determining those vulnerable component processes, mean time between failures, base longevity interval, repair rates, downtime costs due to unexpected failures and due to scheduled rejuvenation, etc., for an application is a difficult and imprecise task. Most applications collect failure data, do RCA (Root-Cause Analysis) of their failures and keep track of their costs; data collected from those reports may provide some insights into the values for those variables. From those values, one can use Equations 6 and 7 to determine whether rejuvenation would be beneficial at all or not. If it is beneficial, one can compute the cumulative downtime and cost due to that downtime using Equations 3 and 4 for a range of rejuvenation rates and pick an appropriate rejuvenation frequency.

Observe that, for the rejuvenation to be beneficial, the condition in Equation 6 should be negative which implies that c_r should be much less than c_f, i.e. the recovery cost after rejuvenation should be much less than that cost after a failure. So, the rejuvenation times and frequencies should be chosen carefully. Rejuvenation time of day, day of the week etc. should be chosen when it is least disruptive to the application and its clients. Those times can perhaps be obtained from the activity intensity profile of an application.

We discussed only the two-step failure model where the application goes from an initial robust state to a failure probable state before it can either fail or be rejuvenated. If this model is not appropriate for an application, i.e. if it is a one-step failure model where the application goes from a non-faulty state to either a failure state or a rejuvenation state in one-step, then

$r_2 = \infty$. Substituting this value in Equations 1 and 3 shows that, no matter how fast rejuvenation procedure is completed, downtime with rejuvenation will always be larger than the downtime without rejuvenation in one-step failure model. Substituting $r_2 = \infty$ in Equation 4 and differentiating it with respect to r_4 yields the threshold condition $c_r < c_f[\frac{\lambda}{\lambda + r_1}]$ for rejuvenation to be beneficial in a one-step failure model. This condition implies that the cost of scheduled rejuvenation has to be much less than the repair cost after an unscheduled failure, for rejuvenation to be beneficial in a one-step failure model. For example, consider Example B in Section 3.3.2 with a one-step failure model and change the failure rate λ to $\frac{1}{24}$, i.e. the mean time between two consecutive failures is one day. In that case, the rejuvenation cost should be less than about 2% of the repair cost for rejuvenation to be beneficial.

Acknowledgments

The authors are grateful to Larry Bernstein for encouraging us to work together on this topic from an analytical as well as a practical viewpoint. We are also thankful to Sampath Rangarajan of Northeastern University who contributed to the observations on rejuvenation thresholds discussed in Sections 3.2, 3.3 and concluding remarks when he was a consultant at AT&T during the summer of 1994. Dave Belanger, John Musa, Phong Vo, Yi-Min Wang and Elaine Weyuker have also reviewed the concepts, analysis and experiences presented in this paper.

References

[1] E. N. Adams, "Optimizing Preventive Service of Software Products" *IBM Jl. Res. Develop.*, Vol. 28, No. 1, pp. 1-14, January 1984.

[2] L. Bernstein, "Innovative Technologies for Preventing Network Outages," *AT&T Technical Journal,* Vol. 72, No. 4, pp. 4-10, July 1993.

[3] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques,* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[4] Y. Huang and C. M. R. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience", *Proceedings of 23rd Intl. Symposium on Fault-Tolerant Computing*, Toulouse, France, pp. 2-9, June 1993. An expanded version of this paper appeared as a chapter titled "Software Fault Tolerance in the Application Layer" in the book, *Software Fault Tolerance*, M. Lyu (Editor), John Wiley & Sons, March 1995.

[5] Y. Huang, P. Jalote and C. M. R. Kintala, "Two Techniques for Transient Software Error Recovery", *Hardware and Software Architectures for Fault Tolerance: Experience and Perspectives*, Edited by M. Banatre and P. A. Lee, Springer Verlag, Lecture Notes in Computer Science, No. 774, pp. 159-170, 1994.

[6] F. Lin, "Re-Engineering Option Analysis for Managing Software Rejuvenation" *Information & Software Technology*, Vol. 35, No. 8, pp. 462-467, August 1993.

[7] J. D. Musa, A. Iannino and K. Okumoto, *Software Reliability Measurement, Prediction, Application*, McGraw Hill, 1987.

[8] D. L. Parnas, "Software Aging", *Proceedings of 16th Intl. Conference on Software Engineering*, Sorrento, Italy, pp. 279-287, May 1994.

[9] D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems Design and Implementation,* Digital Press, 1992.

[10] Y. M. Wang, Y. Huang and W. K. Fuchs, "Progressive Retry for Software Error Recovery in Distributed Systems," *Proceedings of 23rd Intl. Symposium on Fault-Tolerant Computing,* Toulouse, France, pp. 138-144, June 1993.

[11] Y. M. Wang, Y. Huang, K. P. Vo, P. Y. Chung and C. Kintala, "Checkpointing and Its Applications," *Proceedings of 25th Intl. Symposium on Fault-Tolerant Computing*, Pasadena, California, June 1995.

[12] E. J. Weyuker and A. Avritzer, "Estimating Software Reliability of Smoothly Degrading Systems," Proceedings of *5th International Symposium on Software Reliability Engineering*, pp.168-174, November 1994.