A Performability-Oriented Software Rejuvenation Framework for Distributed Applications*

Ann T. Tai Kam S. Tso
IA Tech, Inc.
Los Angeles, CA 90024

William H. Sanders
University of Illinois
Urbana, IL 61801

Savio N. Chau
Jet Propulsion Laboratory
Pasadena, CA 91109

Abstract

While inherent resource redundancies in distributed applications facilitate gracefully degradable services, methods to enhance their dependability may have subtle, yet significant, performance implications, especially when such applications are stateful in nature. In this paper, we present a performability-oriented framework that enables the realization of software rejuvenation in stateful distributed applications. The framework is constructed based on three building blocks, namely, a rejuvenation algorithm, a set of performability metrics, and a performability model. We demonstrate via model-based evaluation that this framework enables error-accumulation-prone distributed applications to deliver services at the best possible performance level, even in environments in which a system is highly vulnerable to failures.

1 Introduction

Software rejuvenation has been recognized as a simple yet effective means for avoiding computer system failures that are caused by accumulative internal errors or error-caused system capacity deterioration. When a software system is voluntarily rebooted, with high probability, errors accumulated at runtime will be eliminated and the system will regain its full capacity. The subject of software rejuvenation has been investigated extensively by researchers (see [1, 2, 3, 4, 5, 6], for example). More recently, research efforts have suggested the use of software rejuvenation to protect critical systems against "incremental intrusions" [7], which are difficult to detect and isolate before damage accumulates to the point that catastrophic failures expected by attackers become inevitable.

Inspired by the prior work and motivated by the fact that rapidly advancing network technologies have resulted in the reliance of an ever-increasing fraction of the world's infrastructure upon distributed software systems, we develop a software rejuvenation framework for stateful distributed applications that comprise server replicas. Such applications represent an important area, but one in which software rejuvenation has not yet been considered. Our goal is to investigate the feasibility of software rejuvenation in stateful distributed applications and to address the design considerations that are necessary for and unique to this important class of distributed systems.

While inherent resource redundancies in distributed applications have the potential to support novel architectures or algorithms and facilitate gracefully degradable performance, methods to enhance their dependability may have subtle, yet significant, performance implications, especially when such applications are stateful in nature. In particular, software rejuvenation which needs to temporarily stop a long-running replica server R may lead to post-rejuvenation performance degradation, since an operation disruption may cause the state of R to become inconsistent with that of other replicas and may imply that R would be unable to provide services at its full capacity until after it restores consistency. Accordingly, we take a performability-oriented approach to developing a framework for the realization of software rejuvenation in stateful distributed applications. By "performability-oriented," we mean that the design, evaluation, and enhancement of the framework are guided by the analysis of a system's ability to deliver services with gracefully degradable performance. The framework is constructed based on three building blocks, namely, a rejuvenation algorithm, a set of performability metrics, and a performability model. The performability metrics and model both take into account the post-rejuvenation performance degradation resulting from consistency restoration. Moreover, our performability model accommodates the fact that the post-rejuvenation consistency restoration processes themselves are not immune to failures, due to the potential performance stress caused by the service requests accumulated during rejuvenation.

The basic version of our rejuvenation algorithm is a precursor-detection-triggered rejuvenation scheme that uses pattern-matching mechanisms to detect pre-failure conditions. To compensate for the inherent limitation of pattern-matching mechanisms, namely, their inability to detect pre-failure condition patterns that are not known a pri-

*The work reported in this paper was supported in part by Small Business Innovation Research (SBIR) Contract NAS3-99125 from the Jet Propulsion Laboratory, National Aeronautics and Space Administration.

ori, the enhanced version of the algorithm accommodates a random timer and allows detection-triggered rejuvenation and timer-triggered rejuvenation to coordinate in a synergistic manner. We demonstrate via model-based evaluation that our performability-oriented framework enables error-accumulation-prone distributed applications to continuously deliver gracefully degradable services at the best possible performance level, even in environments in which a system is highly vulnerable to failures. Further, our effort shows that software rejuvenation can be realized as an integral part of the infrastructure in stateful distributed applications that guarantee *eventual consistency*.

The remainder of the paper is organized as follows. Section 2 describes the system model. Section 3 discusses the fundamental concepts and methods that enable the framework development. Sections 4 and 5 describe and analyze the basic and enhanced versions of our rejuvenation algorithm, respectively. We conclude in Section 6.

2 System Model and Assumptions

We focus on an important class of highly available stateful distributed applications in which servers are replicated at multiple sites that are connected through LANs or WANs. Clients interact with the replicated servers by issuing service requests. A requested service will be a "read," "write," or "read and write" access to a logical entity of data.

Distributed replicated services enable a server to be accessed by multiple clients concurrently and provide significantly better response time than a centralized system would. On the other hand, concurrent access has the potential to cause inconsistency among replicas when the application is stateful. Traditional concurrency control protocols, such as two-phase locking, guarantee strong consistency for those applications. However, those protocols often incur high performance overhead when servicing update requests. Hence, over the past fifteen years, the distributed computing community has gradually shifted its research focus from the traditional, expensive concurrency control protocols to their variants that use more flexible algorithms to guarantee eventual rather than immediate consistency when such guarantees comply with clients' requirements.

Our system model presumes a protocol that guarantees eventual consistency by employing a *sequencer* to ensure that update requests from clients are executed in the same order in all the replicas [8]. Based on the system model, we make two assumptions for the framework design and evaluation as follows. First, we assume that the sequencer also takes responsibility for scheduling time-based rejuvenation. Moreover, the sequencer (which resides on a replica server R) has passive replications at other replica servers; thus, the scheduling function, which we call a *scheduler* for simplicity in the remainder of the paper, can migrate to another replica server site in case of site failure.

Second, we assume that when error accumulation becomes excessive before a replica undergoes rejuvenation, the replica will crash. A local recovery is possible, but is not guaranteed to be successful. If a crashed replica fails to recover locally, the subsequent attempt at recovery must rely on another replica that is operational to transfer necessary state information through the network. Consequently, a crashed replica that does not succeed in the first recovery attempt will not have another chance to recover, unless the system has at least one operational replica. In turn, this implies that when all the replicas become non-operational and none of them succeeds in local recovery, the system operation will cease and must wait for major corrective maintenance involving human intervention.

3 Foundation of the Framework

3.1 Basic Approach

From a strict availability point of view, software rejuvenation could be easily implemented in a distributed system with replicas, because system-level availability would not be affected when a single replica or a proper subset of the replicas undergoes rejuvenation. However, this assumption is feasible only for stateless distributed applications. More specifically, software rejuvenation requires a replica R to be periodically or occasionally taken offline for a brief duration, implying that R's state may become obsolete and thus be inconsistent with the rest of the replicas in the system when R is back from rejuvenation.

On the other hand, the notion of eventual consistency can be exploited to enable the realization of software rejuvenation in stateful distributed applications. In particular, for a system in which a protocol that guarantees eventual consistency is employed, the state of a replica will be permitted to be temporally inconsistent with that of other replicas, facilitating the utility of software rejuvenation. Based on that observation, we derive a simple yet cohesive solution that permits a replica to undergo rejuvenation without violating the eventual consistency property.

More specifically, when a replica R undergoes rejuvenation, read requests issued at the local site to R will be diverted to and processed by an in-service replica, while all the update requests will be received/processed by all the in-service replicas. In the meantime, those update requests and the global sequence number (which is abbreviated as *GSN* and enables the updates to be executed in the same order by all the replicas [8]) assignments broadcast by the sequencer will be saved in a buffer at R and will be processed by R according to the global order after R completes its rejuvenation procedure.

Consequently, a replica R that is engaged in rejuvenation can be regarded as one that is temporarily very slow in executing updates, and its state and the states of all other replicas in the system will converge at a point after R's rejuve-

Table 1: Relationships between Coarse- and Fine-Grained Performance Levels

Coarse-Grained	*Fine-Grained*	*Definition*	*Interpretation*
level-0	level-0.0	$i = 0, j = 0$	None of the n replicas are operational.
level-1	level-$i.j$	$i \in \{1, \ldots, n-1\}, j \in \{0, \ldots, i\}$	A non-empty proper subset of the n replicas are operational.
level-2	level-$n.j$	$i = n, j \in \{0, \ldots, n-1\}$	All the n replicas are operational and a proper subset of them are in-service replicas.
level-3	level-$n.n$	$i = n, j = n$	All the n replicas are operational and are in-service replicas.

nation. Coupled with inherent resource redundancy, this enables stateful distributed applications to deliver gracefully degradable services without eventual-consistency property violation or rejuvenation-caused unavailability.

Finally, the basic approach illustrated above enables us to develop algorithms for software rejuvenation in stateful distributed systems, as described in Sections 4 and 5.

3.2 Performance-Level Definition

According to the discussion in Section 3.1, a client who issues a request to a replica R that is under rejuvenation will perceive a response time that is slower but likely acceptable. Further, since all update requests accumulated in R's buffer will be processed after R completes rejuvenation, and since in the meantime additional service requests may arrive at R, R may have a stressful period with a duration τ following its rejuvenation (see Section 4.2.3 for an analytical discussion). As the state of R is expected to become consistent (to the degree that it would have been if the rejuvenation had not happened) with that of other replicas by the end of τ, we use the term *consistency restoration* to refer to R's services to the accumulated update requests during τ. Moreover, even with the preventive measure, replicas may still crash, and a crashed replica R' will undergo a recovery process during which it will remain non-operational until the recovery completes. Consequently, R' may also experience a stressful period following its recovery for consistency restoration.

The above analysis suggests a pair of index (random) variables whose values will determine the possible performance levels of a system, namely,

I: The number of operational replicas.

J: The number of operational replicas that are not engaged in either rejuvenation or consistency restoration[1].

Then, the performance levels of a system with n replica servers can be expressed in terms of I and J as follows:

$$\{\text{level-}i.j \mid i \in \{0, \ldots, n\}, j \in \{0, \ldots, i\}\} \quad (1)$$

Accordingly, when we say that a system is performing at level-$i.j$, we mean that the system is in a state in which i replicas are operational and among them j are not engaged in either rejuvenation or consistency restoration. Further, if we consider the performance impact from a replica failure to be far more severe than the impact found when one or more operational replicas are undergoing rejuvenation or consistency restoration, then the above mathematical expression not only generates a sequence of performance levels for a given value of n, but also ranks them systematically in an increasing order based on the values of I and J.

In addition, the above expression allows us to choose the "granularity" for defining performance levels. More specifically, we can group the fine-grained performance levels derived from the above expression based on the value ranges of I and J to obtain a set of coarse-grained performance levels for the same system. Table 1 provides an instance of such grouping and shows the relationships between the coarse- and fine-grained performance levels.

In turn, the performance levels so obtained enable us to define a set of performability metrics, a building block that allows us to quantify a system's ability to provide gracefully degradable services. In Sections 4 and 5, we define performability metrics directly in terms of those performance levels, in the context of an instance of our system model that is used to illustrate the basic and enhanced versions of our rejuvenation algorithm.

4 PD Scheme

4.1 Precursor-Detection Approach

Researchers have proposed several software rejuvenation schemes that are based on prediction, observation, and monitoring (see [5, 6], for example). Since those schemes allow rejuvenation to take place only when failures are likely to happen, they reduce the system unavailability caused by rejuvenation activities that are unnecessary.

Accordingly, we begin by proposing a precursor-detection-triggered rejuvenation scheme (PD scheme). The scheme uses patterns of pre-failure conditions that are identified based on earlier data from the system logs that offer clues on the relationships between failures and their preconditions. For example, a rapid, monotonic, or accelerated increase of memory utilization may be identified as a pre-

[1] For brevity, in this paper we sometimes use the term "in-service replicas" to refer to those replicas.

Table 2: Definition and Interpretation of Performance Levels

Level	*Definition*	*Interpretation*
0	$i = 0, j = 0$	Neither of the replicas is operational.
1	$i = 1, j \in \{0, 1\}$	One replica is operational and the other is not.
2	$i = 2, j \in \{0, 1\}$	Both replicas are operational and at least one of them is engaged in rejuvenation or consistency restoration.
3	$i = 2, j = 2$	Both replicas are operational and neither of them is engaged in rejuvenation or consistency restoration.

failure condition if data show that it is strongly correlated with system failures (due to memory leakage, for example). When an observed system event or a combination of observed events matches a pre-defined pattern, an exception will be raised to activate a rejuvenation procedure.

The main advantage of pattern-matching detection mechanisms (which sometimes are called "misuse detection" in the context of intrusion detection systems) is their ability to accurately detect instances of known pre-failure condition patterns with only very limited false alarms. Their disadvantage is the lack of an ability to detect the pre-failure conditions that are not known a priori. There are alternative types of detection mechanism that offer better detection coverage, such as "anomaly detection," which treats observed system behaviors that deviate from the established normal system profiles as "anomalies." Nonetheless, a severe drawback of that mechanism is its detection accuracy problem (i.e., a high rate of false alarm). For software rejuvenation purposes, we choose a high-accuracy detection mechanism, namely pattern matching, over those high-coverage high-false-alarm-rate ones, as they would defeat the purpose of using detection to minimize system unavailability caused by unnecessary rejuvenation activities.

To assess the PD scheme, we conduct a performability evaluation in the following subsection. For simplicity and clarity, we choose to analyze a distributed application that is an instance of our system model with $n = 2$.

4.2 Performability Model for the PD Scheme

4.2.1 Performability Metrics

From the performance-level definition described in Section 3.2, it follows that a system with two replicas can have four coarse-grained performance levels, as shown in Table 2.

The performance-level definition then enables us to define a set of performability metrics as follows:

$$\{P(Y = k) \mid k \in \{0, 1, 2, 3\}\}.$$

Since we are also interested in evaluating the probability that the system will perform at or above a level k, we define a pair of additional metrics $\{P(Y \geq k) \mid k \in \{1, 2\}\}$. Note that values 0 and 3 are excluded from the range of k. The reason is that they correspond to two degenerate cases, namely, $P(Y \geq 0)$ and $P(Y \geq 3)$, which are equal to 1 and $P(Y = 3)$, respectively.

4.2.2 SAN Model

To evaluate the performability metrics, we build a performability model with stochastic activity networks (SANs) using *UltraSAN* [9]. The SAN model for the PD scheme executing on a 2-replica application is depicted in Figure 1. The replicas are both in a robust state initially (represented by tokens in the places `n1_robust` and `n2_robust`). As time passes and error conditions accumulate, a replica will eventually enter a vulnerable state, which is represented by a token in place `n1_vuln` or `n2_vuln`.

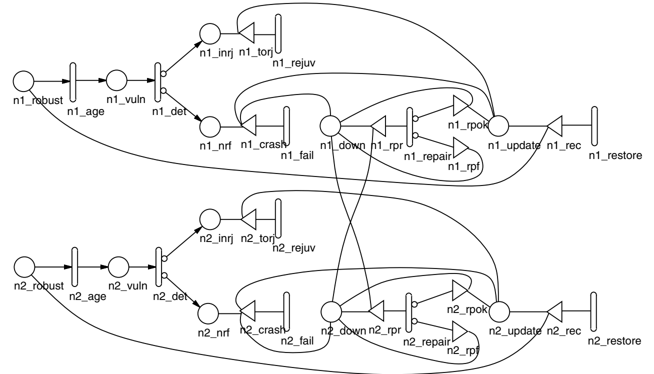


Figure 1: SAN Model for the PD Scheme

Note that an early-stage vulnerable state is generally not detectable[2], which means that the error accumulation may take time to reach a point at which a pattern-matching mechanism is likely to capture the precursor of failure. The transition from a vulnerable state to a detectable near-failure state is represented by the timed activities `n1_det` and `n2_det`. Each of the activities is associated with two cases, indicating the detection mechanism's imperfect coverage. Undetected pre-failure conditions at this stage are likely to manifest themselves quickly into a replica fail-

[2]An attempt to detect early-stage pre-failure conditions may lead to a high rate of false alarm, since a loose pattern match does not necessarily mean a precursor of failure.

ure, which is modeled by the timed activities n1_fail and n2_fail. As all those timed activities have exponential distributions, the time it takes a replica to go from a robust state to a failure state has a hypo-exponential distribution (which is one form of increasing failure rate distribution [6]).

As discussed in Section 3, software rejuvenation in the distributed applications we consider must be followed by consistency restoration. Therefore, upon the completion of the activity n1_rejuv (or n2_rejuv), another timed activity n1_restore (or n2_restore) will be enabled. Note also that there are arcs from the places n1_update and n2_update to the input gates n1_crash and n2_crash (respectively); this enables the model to accommodate the fact that the post-rejuvenation/recovery consistency restoration processes themselves are not immune to failures, due to the potential performance stress caused by the service requests accumulated during rejuvenation or recovery.

In Section 4.2.3, we will describe how we characterize the consistency-restoration duration (for specifying the rates for the activities n1_restore and n2_restore), but we will omit further discussion of the SAN model due to space limitations.

4.2.3 Consistency Restoration Duration

Recall that during a replica's rejuvenation, update requests will be saved to its local buffer and will be processed upon the completion of rejuvenation. Since additional service requests may arrive in the meantime, post-rejuvenation consistency restoration will continue until the completion of a requested service results in an empty buffer and an idle replica server. Hence, the duration of consistency restoration is analogous to a *busy period* [10]. Clearly, the longer a rejuvenation process takes or the higher the update-request rate is, the lengthier the busy period will be. Moreover, the higher server utilization factor during this period is likely to cause a temporary, user-perceived performance degradation. However, the derivation of a busy period probability density function (pdf) is very complex even for the simplest queueing system $M/M/1$ [10]. Although we can compute the first three moments of a busy period and then derive a more accurate exponential pdf approximation based on them [11], in this study we choose to apply an exponential approximation that uses only the first moment, for simplicity. In the following, we derive a solution for the expected value of a busy period, $E[B]$, whose reciprocal will be used as the constant rate in the exponential approximation that characterizes the busy period. Moreover, for clarity, we use the context-specific term "consistency restoration duration" to refer to such a busy period, in the remainder of the paper.

We assume that the update-request interarrival time and service time are both exponentially distributed so that we have an $M/M/1$ queueing system at each replica server. In order to evaluate the expected value of consistency restoration duration $E[B]$ analytically, we view 1) the update requests that have not yet been processed when rejuvenation starts, and 2) the requests accumulated during rejuvenation, collectively, as a single customer C_1. Then C_1 is the customer who initiates a busy period and has a service time distribution different from other customers (i.e., the individual service requests that arrive during consistency restoration). From that perspective, the system becomes an $M/G/1$. Before proceeding to derive the solution for $E[B]$, we define the following notation:

- λ Update request rate.
- μ Update service rate.
- γ Rejuvenation completion rate.
- ω Failure recovery completion rate.
- k_1 Number of update requests that are left in the local system when rejuvenation starts.
- k_2 Number of update requests that arrive at the local system during rejuvenation.
- τ Rejuvenation completion time.
- X_1 Service time of C_1.
- X Service time of C_i, $i \in \{2, 3, \ldots\}$.

Then, if we let S denote the service time of the $M/G/1$ system, we have (see [10], for example)

$$E[B] = \frac{E[S]}{1 - \lambda E[S]}. \quad (2)$$

Due to Poisson arrivals, $a_0 = p_0 = 1 - \lambda E[S]$, where a_0 and p_0 are the steady-state probabilities that an arrival will see an empty system and that the system will be empty, respectively. Thus,

$$E[S] = (1 - \lambda E[S])E[X_1] + \lambda E[S]E[X]. \quad (3)$$

To solve for $E[S]$, we obtain

$$E[S] = \frac{E[X_1]}{1 + \lambda E[X_1] - \lambda E[X]}. \quad (4)$$

Since the service times of individual requests are independently and identically distributed with a constant rate μ (i.e., an exponential distribution), $E[X_1] = kE[X]$, where $k = k_1 + k_2$. Then, by plugging that into Eq. (4), which in turn is plugged into Eq. (2), we obtain an expression for the conditional expectation of consistency restoration duration:

$$E[B \mid k] = \frac{kE[X]}{1 - \lambda E[X]} = \frac{\frac{k}{\mu}}{1 - \frac{\lambda}{\mu}} = \frac{k_1 + k_2}{\mu - \lambda}. \quad (5)$$

Finally, from the theorem of total expectation, it follows that

$$E[B] = \int_{\tau=0}^{\infty} \gamma e^{-\gamma\tau} \sum_{k_1=0}^{\infty} (1-\rho)\rho^{k_1} \sum_{k_2=0}^{\infty} \frac{(\lambda\tau)^{k_2}}{k_2!} e^{-\lambda\tau} \left(\frac{k_1+k_2}{\mu-\lambda}\right) d\tau = \frac{\lambda}{(\mu-\lambda)^2} + \frac{\lambda}{\gamma(\mu-\lambda)} \quad (6)$$

Note that if we replace γ with ω in Eq. (6), we can then compute the expected value of post-recovery consistency restoration duration.

4.3 Quantitative Results

Using the SAN model developed in Section 4.2.2 and applying *UltraSAN*, we conduct parametric studies. Before proceeding to discuss the results, we introduce additional notation as follows:

α: Rate at which a replica enters a vulnerable state from a robust state.

β: Rate at which a replica enters a near-failure state from a vulnerable state.

c_d: Coverage of precursor detection.

η_1: Rate at which a replica enters a crash state from a near-failure state.

η_2: Rate at which a replica enters a crash state from the state in which the replica undergoes consistency restoration.

ν_1: Completion rate of post-rejuvenation consistency restoration.

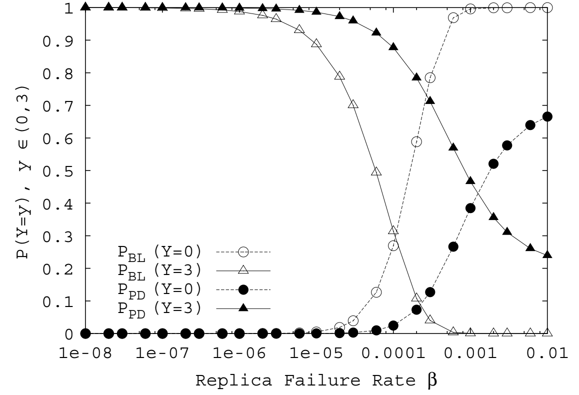
ν_2: Completion rate of post-recovery consistency restoration.

c_{r1}: Probability of a successful recovery from an error-accumulation-caused failure.

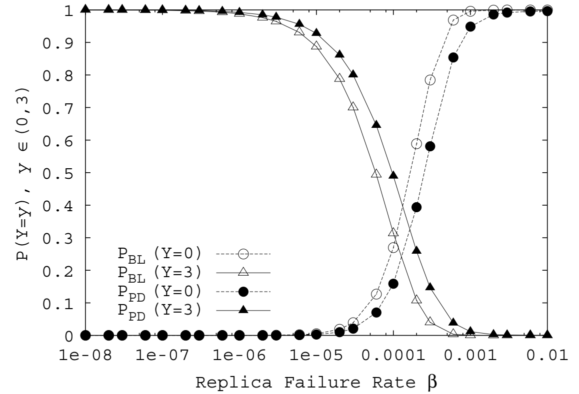
c_{r2}: Probability of a successful recovery from a performance-stress-caused failure.

The first study we conduct is based on the following parameter values (unless noted, all the parameters involving time presume that time is quantified in hours): $\alpha = 0.001$, $c_d = 0.95$, $\eta_1 = 1$, $\eta_2 = 10^{-6}$, $\omega = 2$, $\nu_1 = 4$, $\nu_2 = 1.33$, $\gamma = 6$, $c_{r1} = 0.5$, and $c_{r2} = 0.95$. Note that ν_1 and ν_2 are computed (based on Eq. (6)) as the functions of λ, μ, γ, and ω, with the values of λ and μ set to 150 and 250 (per second), respectively.

Note also that if we set c_d to zero, the performability model shown in Figure 1 will degenerate into a baseline model in which no software rejuvenation will ever be performed. In order to assess the effectiveness of the PD scheme, we also compute the same performability measures for the baseline system. The measures are computed by the transient analytic solver in *UltraSAN* at the time point $t = 10,000$ hours. The evaluation results are illustrated in Figure 2(a), where the results of $P(Y = 0)$ and $P(Y = 3)$ for the PD scheme and baseline (BL) system are compared.



(a) $c_d = 0.95$



(b) $c_d = 0.4$

Figure 2: Improvement from the Use of the PD Scheme

The curves show that after β increases to 10^{-5}, the system with the PD scheme is much more likely to operate at performance level 3 than the baseline system. Meanwhile, the former is much less likely than the latter to be at performance level 0 after β reaches 10^{-5}.

Nonetheless, we get a quite different picture, as shown in Figure 2(b), when c_d is reduced to 0.4. In particular, the improvement from the use of the PD scheme can be observed only in a limited value range of β, and the improvement becomes increasingly negligible when β becomes greater than 0.001. As explained in Section 4.1, while the detection mechanisms using pattern matching have better accuracy

(i.e., fewer false alarms), they tend to have lower detection coverage due to their inability to recognize pre-failure conditions that are not known a priori. In addition, distributed applications nowadays tend to evolve quickly, which may also prevent us from recognizing enough precondition patterns of possible failures. Those factors together suggest to us an approach for enhancing the rejuvenation framework, as described in the next section.

5 PDRT Scheme

5.1 Algorithm

Based on the findings discussed in Section 4.3, we enhance the framework by including in it a random-timer-triggered rejuvenation scheme to complement the precursor-detection-triggered scheme for better performability. In order to let the two schemes coordinate synergistically, the enhanced scheme employs a scheduler which can be implemented as a function of the sequencer in our system model (see Section 2). The duties of the scheduler include 1) keeping track of the detection-triggered-rejuvenation events and failure-caused-recovery events, and 2) determining, upon every timer expiration, which of the replicas should undergo rejuvenation. With the resulting *precursor-detection-triggered random-timer-triggered* (PDRT) rejuvenation algorithm, a replica R will undergo a software rejuvenation upon the detection of a failure precursor or the expiration of the random timer that signals R's turn for rejuvenation, whichever occurs first. Consequently, in the absence of detectable pre-failure conditions, the PDRT scheme will reduce to a solely time-based round-robin rejuvenation policy.

We choose a random timer mainly because random scheduling for preventive maintenance is often used in industry (see [12], for example). To implement the random-timer-triggered, round-robin scheduling policy, we begin by determining a value ϕ for mean time between rejuvenation, or mean *rejuvenation interval*. To let the rejuvenation intervals for all the n replicas have identical distributions (with a mean ϕ), we let the time between two consecutive rejuvenations that are performed by two different replicas have an exponential distribution and have a mean φ which equals $\frac{\phi}{n}$. Finally, to distinguish it from ϕ, we call φ the "mean rejuvenation sub-interval." Consequently, $\phi = n\varphi$ and the rejuvenation interval has a hypo-exponential distribution.

The algorithm is shown in Figure 3. Note that the algorithm allows a choice of $N_{\min}$, which is the minimum number of operational replicas below which the timer-countdown will be disabled so that timer-triggered rejuvenation will not take place (to ensure service availability). We can also see from Figure 3 how the value of the next rejuvenation sub-interval is assigned, how the timer is set, and how k, the ID of the replica that is supposed to undergo rejuvenation upon the next timer expiration, is updated to enable the round-robin scheduling.

```
// n replicas in a networked system
replicaSet = {R_1, R_2, ··· , R_n-1, R_n};
// φ is a replica's mean rejuvenation interval
ϕ = φ/n;
δ = 1/ϕ;
edist = ExponentialDistribution(δ);
τ = getNextIntvl(random(edist));
setTimer(rejuvTimer, opSize(replicaSet), N_min, τ);
k = 1;
while (rejuvEnabled) {
  if (rejuvTimer(opSize(replicaSet), N_min) == 0) {
    if (k ∉ {S_fID, S_rID})
      send(rejuvReqst, k);
    τ = getNextIntvl(random(edist));
    setTimer(rejuvTimer, opSize(replicaSet), N_min, τ);
    update(S_rID, n);
    k = (k mod n) + 1;
  }
}
```

Figure 3: PDRT Algorithm

Combining two different schemes into a framework creates several design issues. In particular, it would not be reasonable to let the scheduler send a replica R to a timer-triggered rejuvenation soon after R completes a detection-triggered rejuvenation or a failure recovery. Rather, in such a scenario, the replica should be allowed to skip the timer-triggered rejuvenation until after a complete rejuvenation cycle (whose ending point is signaled by the nth timer expiration[3] relative to the point at which R completes a detection-triggered rejuvenation or recovers from failure).

The above rule, and a rule that prevents a rejuvenation request from being sent to a crashed replica, are implemented in the PDRT algorithm using two sets that are maintained by the scheduler, $S_{\mathtt{fID}}$ and $S_{\mathtt{rID}}$, which keep track of the replicas 1) that are crashed or engaged in a detection-triggered rejuvenation, and 2) that have just recovered from failure or have just completed a detection-triggered rejuvenation, respectively. More specifically, when a replica R crashes or undergoes a detection-triggered rejuvenation, its ID will be entered into $S_{\mathtt{fID}}$; when R has recovered from failure or completed a detection-triggered rejuvenation, its ID will be transferred from $S_{\mathtt{fID}}$ to $S_{\mathtt{rID}}$. The ID will remain in $S_{\mathtt{rID}}$ until the nth timer expiration (relative to the point when R's ID is transferred into $S_{\mathtt{rID}}$), meaning that a replica will be exempted from rejuvenation in the next rejuvenation cycle after failure recovery or detection-triggered rejuvenation.

Accordingly, as shown in Figure 3, when the timer expires, the scheduler will first check if a replica R that is picked up by the round-robin scheduling policy belongs to $S_{\mathtt{fID}}$ or $S_{\mathtt{rID}}$, and will send a rejuvenation request to R if and only if R is neither in $S_{\mathtt{fID}}$ nor in $S_{\mathtt{rID}}$.

[3] This does not mean that R has to undergo rejuvenation upon the nth timer expiration, but that R will be allowed to do so when the $(n+k)$th timer expiration ($k \geq 0$) indicates that it is R's turn for rejuvenation according to the round-robin scheduling.

5.2 Performability Model for the PDRT Scheme

We build a SAN model for the PDRT scheme by extending the PD model, as shown in Figure 4. The extension includes the timed activity `rdm_timer`, the input gate `scheduler`, and two places `inop` and `scheduled` whose markings indicate, respectively, the number of operational replicas and the ID of the replica that is scheduled for rejuvenation at the next epoch.

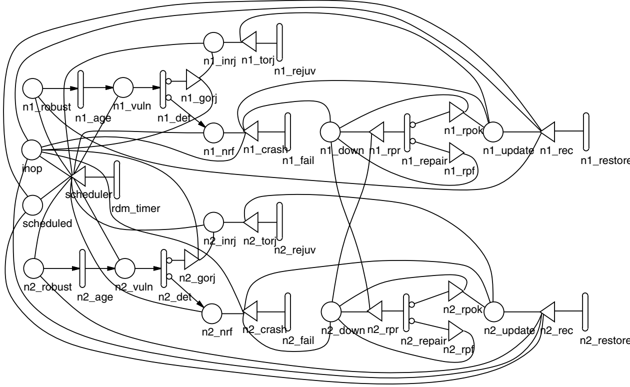


Figure 4: SAN Model for the PDRT Scheme

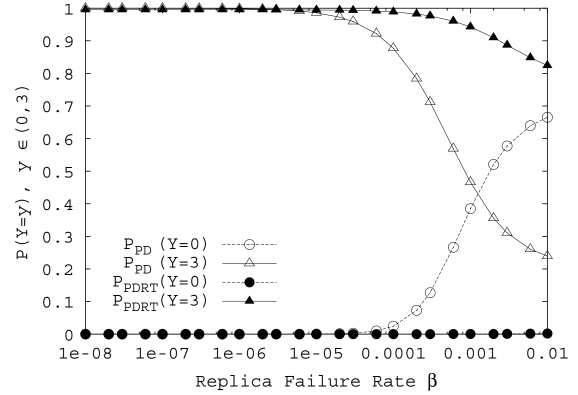
To this end, we obtain a comprehensive model that allows us to evaluate not only the PDRT scheme but also its variants. More succinctly, by assigning a value to $N_{\min}$ such that $N_{\min} > n$, setting c_d to zero, or both, the model will be reduced to a representation of, respectively, the PD scheme, the random-timer-only scheme, or the baseline system. Due to space limitations, we omit further discussion of the model.

5.3 Quantitative Results

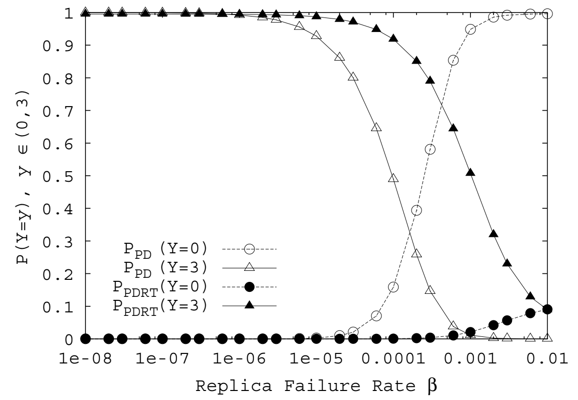
Based on the SAN performability model, we carry out a comparative study for the PD and PDRT schemes by applying the same parameter values we used for the first study presented in Section 4.3. Again, the measures are computed by the transient analytic solver at the time point $t = 10,000$ hours. As shown in Figure 5(a), throughout the domain of β we consider, the use of PDRT always yields a higher value of $P(Y = 3)$ and a lower value of $P(Y = 0)$, relative to PD. We then repeat the study for the case $c_d = 0.4$, and the results are displayed in Figure 5(b).

The curves in Figure 5(b) demonstrate that use of timer-triggered rejuvenation to compensate for the shortcoming of PD, namely its inadequate detection coverage, is effective in reducing the likelihood that the system will be unable to deliver any services at t (i.e., $P(Y = 0)$). In particular, the improvement is very impressive when β becomes greater than 0.0001. On the other hand, the improvement reflected by the results of $P(Y = 3)$ is less significant.

As we note from Figure 5(b) that the summation of $P(Y = 3)$ and $P(Y = 0)$ for PDRT is appreciably less



(a) $c_d = 0.95$



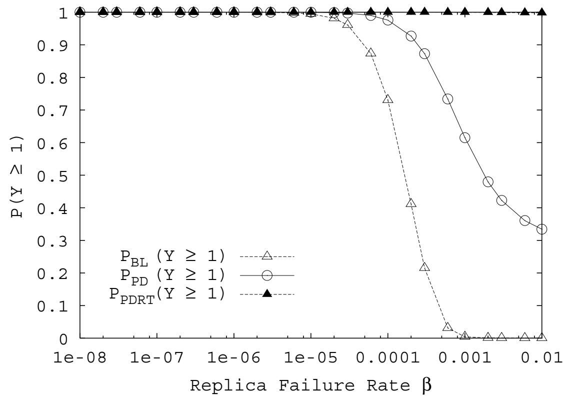
(b) $c_d = 0.4$

Figure 5: Improvement from the Use of the PDRT Scheme

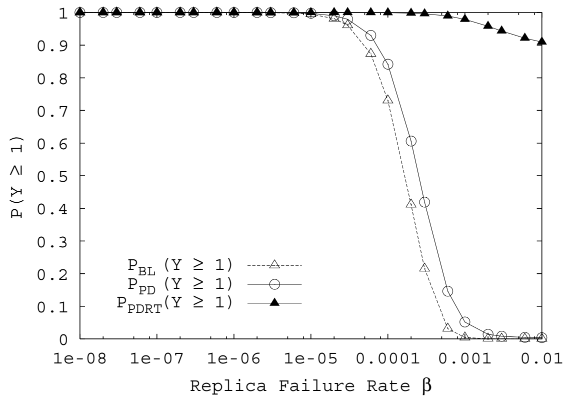
than 1 in the region where β is greater than 0.0001, we further examine the system behavior under PDRT based on the other form of performability measure, namely, $P(Y \geq 1)$. The numerical results are illustrated in Figures 6(a) and 6(b). The data show that with PDRT the system is virtually guaranteed to provide services at performance level 1 or higher if $c_d = 0.95$, while $P(Y \geq 1)$ drops rapidly when β becomes larger than 0.0001 if the system is equipped only with PD or has neither of the rejuvenation schemes.

Also, as shown in Figure 6(b), with $c_d = 0.4$, the improvement from the use of the PD scheme relative to the baseline system becomes increasingly negligible as β approaches 0.01. However, when equipped with PDRT, the system's ability to perform at level one or higher is robust. Tables 3(a) and 3(b) provide us with a more detailed view of how this ability is composed.

We can observe from the tables that even with a high failure rate, a system with PDRT has the best ability to provide services with gracefully degradable performance. Moreover, while a high failure rate and a low detection coverage together are very likely to prevent a system from performing at the highest level, with PDRT the system will still be able



(a) $c_d = 0.95$



(b) $c_d = 0.4$

Figure 6: Gracefully Degradable Performance

Table 3: $P(Y \geq 1)$ and Its Composition ($\beta = 0.01$)

Scheme	$P(Y = 1)$	$P(Y = 2)$	$P(Y = 3)$	$P(Y \geq 1)$
BL	2.13e-04	1.89e-09	2.94e-06	0.00022
PD	9.43e-02	4.73e-04	2.39e-01	0.33423
PDRT	1.69e-01	5.23e-03	8.24e-01	0.99796

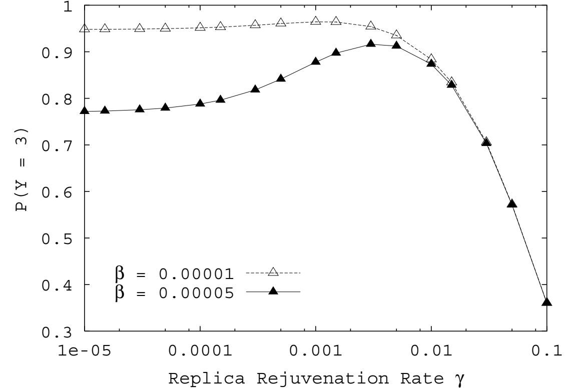
(a) $c_d = 0.95$

Scheme	$P(Y = 1)$	$P(Y = 2)$	$P(Y = 3)$	$P(Y \geq 1)$
BL	2.13e-04	1.89e-09	2.94e-06	0.00022
PD	3.26e-03	4.89e-07	4.01e-04	0.00366
PDRT	8.20e-01	4.98e-04	8.96e-02	0.90959

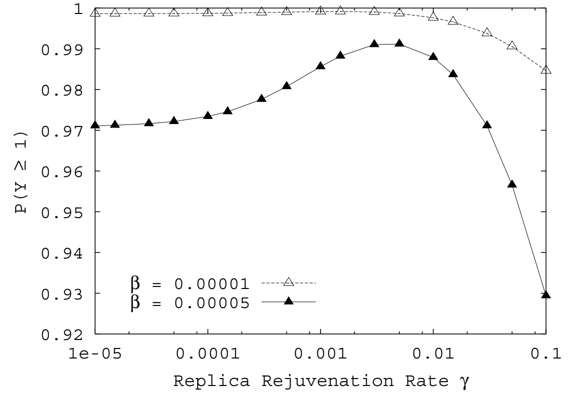
(b) $c_d = 0.4$

to perform at level one with a high probability. In particular, as shown in Table 3(b), even when β and c_d are equal to 0.01 and 0.4, respectively, PDRT will still be able to perform at level one with a probability greater than 80%. In turn, this enables PDRT to perform at level one or higher with a probability greater than 90% for that extreme case, as shown in Table 3(b).

Next we conduct evaluation to study the optimal rejuvenation rate (i.e., the reciprocal of the mean rejuvenation interval ϕ) for the PDRT scheme. Figure 7(a) shows the results of the optimal rate with respect to the system's ability to deliver services at the highest performance level. The curves show that the optimal rates are 0.001 and 0.003 (equivalent to mean rejuvenation intervals of 1,000 hours and 333 hours, respectively), for the cases in which β equals 10^{-5} and 5×10^{-5}, respectively.



(a) $P(Y = 3)$ as the objective function



(b) $P(Y \geq 1)$ as the objective function

Figure 7: Optimal Rejuvenation Rates

We then examine the optimal rejuvenation rates with respect to the system's ability to perform at level one or higher. The results are shown in Figure 7(b), from which we can observe that the optimal rejuvenation rates become 0.0015 and 0.005 (equivalent to mean rejuvenation intervals of 666 hours and 200 hours, respectively). Note that the optimal rates are higher than those when $P(Y = 3)$ is used as the objective function. The reason is that the objective function $P(Y \geq 1)$ represents a less strict criterion that permits the system to undergo rejuvenation more frequently to minimize the likelihood that the system will be in a state corresponding to performance level zero.

Finally, as shown in Eq. (6), consistency restoration duration is a non-linear increasing function of update-request rate λ, which means that a small increase of λ may cause the

post-rejuvenation performance degradation to linger much longer. In turn, this suggests that our algorithm can be enhanced by inclusion of a workload-adaptive mechanism that will stop the timer (or decrease the rejuvenation rate) when the update-request rate exceeds a threshold, so that software rejuvenation can be performed solely on the precursor-detection basis in that situation. Accordingly, our plan for subsequent work includes a case study that will investigate the feasibility of the framework and the adaptive approach explained above, based on a networked space data system for future planetary exploration missions.

6 Concluding Remarks

We have developed a performability-oriented software rejuvenation framework for stateful distributed applications, which is an important area, but one that has not yet been explored for the utility of software rejuvenation. We demonstrate that our framework enables error-accumulation-prone distributed applications to continuously deliver services that are gracefully degradable.

As noted earlier, the intent of this paper is not just to show a specific approach to software rejuvenation that is superior to other rejuvenation schemes or policies. Rather, the emphasis of the paper is on exploring the possibility of using software rejuvenation in stateful distributed applications and on addressing the design considerations that are necessary for and unique to this important class of distributed applications. Accordingly, while our effort is greatly inspired by prior research work on software rejuvenation, we go one step further toward generalizing and enriching its concepts, methods, application domain, and design space.

Furthermore, in contrast to the previous efforts in which mathematical modeling was usually the focus, the major building block of our framework is an algorithmic approach to the realization of software rejuvenation in distributed computing environments. In addition, the set of performability metrics that have guided our framework development and enhancement are directly defined on performance levels and evaluated using an unweighted reward rate, which enables explicit assessment of a system's ability to provide services with gracefully degradable performance.

In summary, the significance of this investigation is twofold. First, our framework is the first contribution to the realization of software rejuvenation in stateful distributed systems, and can be utilized to enable distributed applications to deliver gracefully degradable services at the best possible performance level, even in situations in which the system is highly vulnerable to failures. Second, our novel application of the concept of eventual consistency allows software rejuvenation to be carried out in stateful distributed computing environments without causing service unavailability or sacrificing data consistency. Indeed, we have exploited a particular synergy between the techniques for distributed computing and dependable computing, which may enable or facilitate the realization (in stateful distributed systems) of other state-of-the-art dependability enhancement methods that require occasional or periodical brief stopping of one or more server replicas, such as secure reboot, recovery-oriented computing, self-healing algorithms, and beyond.

References

[1] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Digest of the 25th Annual International Symposium on Fault-Tolerant Computing*, (Pasadena, CA), pp. 381–390, June 1995.

[2] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, "Analysis of preventive maintenance in transaction based software systems," *IEEE Trans. Computers*, vol. 47, pp. 96–107, Jan. 1998.

[3] A. T. Tai, L. Alkalai, and S. N. Chau, "On-board preventive maintenance: A design-oriented analytic study for long-life applications," *Performance Evaluation*, vol. 35, pp. 215–232, June 1999.

[4] T. Dohi, K. Goseva-Popstojanova, and K. S. Trivedi, "Estimating software rejuvenation schedules in high assurance systems," *Computer Journal*, vol. 44, no. 6, pp. 473–485, 2001.

[5] A. Bobbio, M. Sereno, and C. Anglano, "Fine grained software degradation models for optimal rejuvenation policies," *Performance Evaluation*, vol. 46, pp. 45–62, Sept. 2001.

[6] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi, "Analysis and implementation of software rejuvenation in cluster systems," in *Proceedings of SIGMETRICS 2001*, (Cambridge, MA), pp. 62–71, June 2001.

[7] A. Valdes *et al.*, "An architecture for an adaptive intrusion tolerant server," in *Proceedings of the Security Protocols Workshop*, (Cambridge, UK), pp. 158–178, Apr. 2002.

[8] S. Krishnamurthy, W. H. Sanders, and M. Cukier, "An adaptive quality of service aware middleware for replicated services," *IEEE Trans. Parallel and Distributed Systems*, vol. 14, pp. 1112–1125, Nov. 2003.

[9] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko, "The *UltraSAN* modeling environment," *Performance Evaluation*, vol. 24, no. 1, pp. 89–115, 1995.

[10] L. Kleinrock, *Queueing Systems. Volume I: Theory*. New York: John Wiley & Sons, 1975.

[11] J. Abate and W. Whitt, "Limits and approximations for the busy-period distribution in single-server queues," *Probability in the Engineering and Informational Sciences*, vol. 9, pp. 581–602, 1995.

[12] S.-H. Sheu and W. S. Griffith, "Optimal age-replacement policy with age-dependent minimal-repair and random-leadtime," *IEEE Trans. Reliability*, vol. 50, pp. 302–309, Sept. 2001.