Computing (2014) 96:503–526
DOI 10.1007/s00607-014-0385-x

Software rejuvenation in cluster computing systems with dependency between nodes

Menghui Yang · Geyong Min · Weikang Yang · Zituo Li

Received: 11 February 2013 / Accepted: 23 January 2014 / Published online: 17 March 2014

Abstract Software rejuvenation is a preventive and proactive fault management technique that is particularly useful for counteracting the phenomenon of software aging, aimed at cleaning up the system internal state to prevent the occurrence of future failure. The increasing interest in combing software rejuvenation with cluster systems has given rise to a prolific research activity in recent years. However, so far there have been few reports on the dependency between nodes in cluster systems when software rejuvenation is applied. This paper investigates the software rejuvenation policy for cluster computing systems with dependency between nodes, and reconstructs an stochastic reward net model of the software rejuvenation in such cluster systems. Simulation experiments and results reveal that the software rejuvenation strategy can decrease the failure rate and increase the availability of the cluster system. It also shows that the dependency between nodes affects software rejuvenation policy. Based on the theoretic analysis of the software rejuvenation model, a prototype is implemented on the Smart Platform cluster computing system. Performance measurement is carried out on this prototype, and experimental results reveal that software rejuvenation can effectively prevent systems from entering into disabled states, and thereby improving the ability of software fault-tolerance and the availability of cluster computing systems.

This work was supported in part by National Natural Science Foundation of China under the grant No. 60872044, 71133006, and Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China .

M. Yang (✉)
Electronic Records Management Research Center, School of Information Resource Management, Renmin University of China, Beijing 100872, China
e-mail: yangmenghui@ruc.edu.cn

G. Min
Department of Computing, University of Bradford, Bradford BD7 1DP, UK

W. Yang · Z. Li
Research Institute of Information Technology, Tsinghua University, Beijing 100080, China

Keywords Software aging · Software rejuvenation · Cluster computing systems · Stochastic reward net

Mathematics Subject Classification 68N01

1 Introduction

Unplanned computer system outages are more likely to be the result of software failure than of hardware failure. 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 phenomenon called software aging, and may be caused by errors in the application, middleware, or operating system. Since the notion of software aging was first introduced in [1,2], it has attracted increasing interests from both academia and industry. Software aging is an appearance observed in a software application executing continuously for a long period of time, where the state of software degrades and leads to performance degradation or failure. Eventually, this may lead to performance degradation of the software or crash/hang failure or both. Some common causes of software aging are memory bloating and leaking, unreleased file-locks, data corruption, storage space fragmentation, and accumulation of round-off errors. The occurrence of software aging in real systems has been documented in literature [3–5]. Several approaches have been used to study this appearance. The main research efforts have concentrated on the aging effects and their effective mitigation. Moreover, the search for mitigation resulted in the software rejuvenation techniques. Software rejuvenation was first proposed in [2]. This counteracts the aging phenomenon in a proactive manner by removing the accumulated error conditions and freeing up operation system resources. The basic idea is to pause or halt the running software, refresh its internal states, and resume or restart it. Software rejuvenation can be performed by relying on a variety of indicators of software aging, or on the time elapsed since the last software rejuvenation operations. Software rejuvenation can occur at any of the following scopes: system, application, process, or thread. The entire software system can be restarted or the tiniest part of an execution flow can be restarted [6].

Two main categories can be identified among these works, that is, model-based analysis and measurement-based analysis. In model-based analysis, a mathematical model of the system is considered, that includes states in which the system is correctly working, states in which the system is failure-prone, and states in which software rejuvenation is taking place. Several kinds of model have been considered for this purpose, such as Markov processes and stochastic Petri Net. These studies typically aimed at identifying the optimal time for applying software rejuvenation strategies in order to maximize availability or performability in long term. Measurement-based analysis is instead based on data collected from a real system about resource usage (e.g., free physical memory and used swap space) and performance (e.g., response throughput and latency). These data are processed using algorithms for time series analysis [3] and machine learning [7], in order to identify resource exhaustion/performance degradation trends. One of the aims of these studies is to provide a support to online planning of software rejuvenation, in order to predict aging failures in the short

term and to adapt rejuvenation to the actual workload and resource consumption of the system. Another aim of measurement-based analysis is to provide empirical data about the software aging phenomenon, and these data could then be used to populate mathematical models.

Software rejuvenation is a preventive and proactive fault management technique that is particularly useful for counteracting the appearance of software aging, aimed at cleaning up the system internal state to prevent the occurrence of future failure. The increasing interest in combing software rejuvenation in cluster systems has given rise to a prolific research activity in recent years. However, the existing works on software rejuvenation in cluster systems have never explicitly modeled the dependency between nodes. In these models, nodes in cluster systems are assumed to be statistically independent. However, this assumption of independence doesn't hold in real systems. It is more realistic to consider some form of statistical dependence among nodes in a real system. Neglecting such nature may result in straightforward but inaccurate predictions. This paper investigates software rejuvenation policy in cluster computing systems with dependency between nodes, and reconstructs an stochastic reward net model for the software rejuvenation in cluster computing systems which have dependency between nodes. On the dependency issue, Dugan et.al. [8] studied the dependence of a fault-tolerant system. Gokhale et.al. [9] investigated the dependency characterization of the path-based technique for architecture-based reliability prediction of the software. Popstojanova et.al. [10] researched the failure correlation and its effects on the software reliability measures. Fan et.al. [11] analyzed the dispatch–worker based cluster systems and proposed a SRN model for dispatch–worker based architecture with prediction-based rejuvenation. When compared to cluster systems with flat architecture in which all the nodes share the same functions, dispatch–worker based architecture identified a special dependency relations between nodes.

In this paper, we investigate software rejuvenation policy in cluster computing systems with dependency between nodes, and analyze the system model to derive the optimal rejuvenation policy. The aim is to explore how nodes dependence can be represented in stochastic petri net models, and how nodes dependency affects the software rejuvenation policy. Specifically, we first analyze cluster computing systems with dependency between nodes. Then, we introduce the classification of nodes, and present the necessary and sufficient condition of two nodes belonging to the same category. In order to facilitate this dependency, an SRN model is proposed for the cluster computing systems. Compared to the special dependency relationship between dispatch and worker nodes in [11], in which the dispatch node used a model while the worker nodes used another different model in order to distinguish different nodes, our model uses the same model to all the depended and depending nodes. We consider that the dependency relation between nodes is common to all cluster system, and it is needed to include dependency relations in the model explicitly. Based on this SRN model, we analyze the impact of dependency relation on software rejuvenation policy. Experiment results show that dependency between nodes has impacted on software rejuvenation policy. According to the theoretic analysis of the proposed software rejuvenation model, a prototype is implemented on the Smart Platform cluster computing system. Performance measurement is also carried out on this prototype. Experimental results show that software rejuvenation can effectively prevent systems from entering into disabled states.

The remainder of this paper is organized as follows. Section 2 discuses previous related works. Section 3 investigates the models of software rejuvenation in cluster computing systems with dependency between nodes. Section 4 evaluates the validation of SRN model proposed in Sect. 3 through simulation. Section 5 describes the implementation of software rejuvenation in the cluster computing system Smart Platform, and the rejuvenation performance is tested. Finally, Sect. 6 concludes this paper.

2 Related works

Software rejuvenation is a periodic, preemptive restart of a running system to prevent future failures. It is one aspect of a self-healing system. Two policies have been studied to apply the software rejuvenation: (1) by scheduling periodic actions for rejuvenation; (2) by estimating the time for resource exhaustion and performing some technique for proactive rejuvenation. While the first policy is simple to understand and apply, it does not provide the best result in terms of availability and cost, since it may trigger unnecessary rejuvenation actions. Prediction and proactive rejuvenation is potentially a better option. There are two basic approaches to apply the prediction for software rejuvenation: (a) analytic-based approach; (b) measurement-based approach. While all the analytical approaches were based on the assumption that the rate of software aging was known, in the measurement base approach, the basic idea was to monitor and collect data on the attributes responsible for determining the health of the executing software.

The analytic-based approach uses analytic modeling of a system, assuming some distribution for failures, workloads and repair-time, and tries to obtain the best optimal rejuvenation schedule. A simple and useful model based on continuous-time Markov chains was introduced in [2] to analyze the reliability improvements due to software rejuvenation. This model was then extended using stochastic Petri Nets to study rejuvenation using the fail-over mechanisms in IBM's cluster based systems [12]. In [13], fine grained software degradation models, where one could identify the current degradation level based on the observation of a system parameter, were considered, and optimal rejuvenation policies based on a risk criterion and an alert threshold were presented. Software rejuvenation models based on semi-Markov processes were present in [14]. The models were analyzed for optimal rejuvenation strategies based on cost as well as steady-state availability. Models which used Markov regeneration theory were proposed in [15]. In these models, generally distributed failure time, and the service time rate being an arbitrary function of time, were allowed. In [16], a nonhomogeneous Markov chain was used to establish an hierarchical model for the analysis of proactive fault management in the presence of system resource leaks. In [17], a client–server system was modeled using a cyclic nonhomogeneous Markov chain to capture the variation of the arrival and service rates during a day period in order to optimize resource availability through software rejuvenation.

In the measurement-based approach, the system is monitored during execution. This approach applies statistical analysis to data collected from real systems, and utilizes trend analysis or other techniques to determine a window of time over which to perform rejuvenation in order to prevent unplanned outages. In [18], a methodology of measurement-based was presented for detection and estimation trends and times

to exhaustion of the operation system's resources due to software aging in a Unix operation system. An extension of this work in [19] considered time-varying workload, and a semi-Markov reward model was proposed to estimate the rate of resource exhaustion. Another extension of this work was presented in [20], and a comprehensive model of a measurement-based semi-Markov model was built to derive optimal rejuvenation schedules that maximize availability or minimize downtime cost. A study about software aging in a web server was presented in [3]. Proactive detection of software aging for large online transaction processing servers was studied in [21] using monitoring data collected during a period of 5 months. The collected data were used to train a pattern-recognition tool, and the tool was able to predict the occurrence of software aging with a long time in advance. Another related study was presented in [22]. An multivariate state estimation technique (MSET) method was applied for proactive online detection of dynamic sensor and signal anomalies in nuclear power plants and space shuttle main engine telemetry data in cluster systems. In [23], an approach for software rejuvenation based on automated self-healing techniques was presented. Software aging and transient failures were detected through continuous monitoring of system data and performability metrics of the Application Server. The techniques described have been test with a set of open-source Linux tools and the XEN virtualization middleware. Matias et.al. [24] presented a systematic approach to accelerate the software aging manifestation to reduce the experimentation time, and to estimate the lifetime distribution of the investigated system.

Software rejuvenation has been implemented in serval real life systems. The AT&T billing applications [2] and telecommunications switching software [25] both had some form of rejuvenation implementation. Software rejuvenation was also proposed for cable and DSL modem gateway in cable modem terminal system[26] for failure detection and prevention. Software rejuvenation was implemented in specialized transaction processing system [21]. The popular web server software, Apache, implements a form of rejuvenation by killing and recreating processes after a certain numbers of requests have been served [4]. Preventive maintenance, to maximize the probability of successful mission completion, was proposed for spacecraft systems [27]. In [28], a new technique for fast rejuvenation of virtual machines(VM) called warm-VM reboot was presented. It enabled efficiently rebooting only a VM by suspending and resuming it without saving its memory images to persistent storage. In a cluster environment, the warm-VM reboot achieved higher total throughput than the system using VM migration and a normal rebooting.

More recently, two kinds of rejuvenation policies, i.e., the periodic policy and the prediction-based policy, have been implemented in cluster systems to improve performance and availability by taking advantage of the failover feature [12,29]. In the periodic policy, rejuvenation of the cluster nodes was done in a rolling fashion after a very deterministic interval. In the prediction-based policy, the time to rejuvenation was estimated based on the collection and statistical analysis of system data.

3 Models and analysis

In this section, we provide the details of the software rejuvenation in cluster computing systems with dependency between nodes. Specifically, we first introduce the

dependency between nodes into cluster computing systems. Then we present the SRN sub-model of a category U_i in the cluster system. Finally, software rejuvenation is introduced into the proposed model and different software rejuvenation strategies are discussed in this model.

3.1 Cluster computing systems with dependency between nodes

A cluster computing system uses commercially available computers networked in a loosely-coupled fashion, and can provide high level reliability if appropriate levels of error detection and recovery software are implemented in the middleware and application layer. We consider a cluster computing system with n active processors $P_1, P_2, \ldots, P_n$, where n is 2 or greater. In this system there is a power dog attached to each processor that can power on or power down the processor, and a watch dog with connections to each processor that monitors heartbeats from each processor and initiates failover if it detects a processor failure. Given a computer system S, if S meets the following conditions, we call S is a cluster computing system with dependency between nodes.

Condition 1: S includes n nodes, and $n > 1$;
Condition 2: There exists dependency relations between different nodes, and we use function $R(node_x, node_y)$ describing dependency relation between $node_x$ and $node_y$. For clarity, we can define $R(node_x, node_y) = 1$ if there is dependency relation between $node_x$ and $node_y$. Otherwise, $R(node_x, node_y) = 0$.
Condition 3: Dependency relation is directional.

$$\begin{aligned} &If\ \exists node_x, node_y \in S, and\ x \neq y \\ &then\ R(node_x, node_y) \neq R(node_y, node_x) \end{aligned} \tag{1}$$

Condition 4: Each node should has at least one dependency relation with other nodes.

$$\begin{aligned} &\forall node_i,\ \exists node_y \in S, and\ i \neq y \\ &R(node_i, node_y) \neq 0 \\ &And\ \forall node_x \in S, R(node_x, node_x) = 0. \end{aligned} \tag{2}$$

Condition 5: There is no cyclic dependency.

$$\begin{aligned} &\forall node_i, node_{i+1}, \ldots, node_j \in S, (i < j) \\ &R(node_j, node_i) \prod_{k=i}^{j-1} R(node_k, node_{k+1}) = 0 \end{aligned} \tag{3}$$

Some comments are added for the above conditions: First, 'node' is a broad concept. A 'node' refers to an active processor. Generally, a computer is regarded as a node. Second, the dependency relation indicates that a node works normally on condition that one or many other depended nodes are robust. Dependency function

$R(node_i, node_j) > 0$ refers that $node_i$ is dependent on $node_j$. In other words, $node_i$ works properly on condition that $node_j$ works normally. Finally, in some special cluster systems, there are not obvious dependencies between nodes, or cyclic dependencies exist. In this case, we can split a particular node into two nodes to destroy the cyclic dependency relations.

For a cluster computing system S with n nodes, we divide n nodes into m categories, $U_0,\ U_1, \ldots,\ U_{m-1}$, and $U_i \neq \phi$ $(i = 0, \ldots, m-1)$. Each node can only belong to a certain category. In other words,

$$\begin{aligned} &\forall U_i,\ U_j,\ i, j = 0, 1, 2, \ldots, m-1 \\ &if\ i \neq j, then\ U_i \cap U_j = \phi \\ &\cup_{i=0}^{m-1} U_i = S. \end{aligned} \tag{4}$$

There are two rules to determine whether two nodes belong to the same category. One is based on the node's characteristics or behaviors. For two nodes, if they process different affairs, or act as different roles in a task, they are divided into different categories. Here we define a function $F(node)$ to describe node's characteristics or behaviors. For two nodes, if they process different affairs, or act as different roles in a task, they are divided into different categories. Nodes with different functions have different $F(node)$ value. This category rule is identified as follows.

$$\begin{aligned} &\forall node_x, node_y \in S \\ &if\ \exists k,\ and\ node_x \in U_k,\ node_y \in U_k \\ &then\ F(node_x) = F(node_y). \end{aligned} \tag{5}$$

Nodes with different characteristics or behaviors belong different categories U, thus have different $F(node)$ values. For clarity, we can define $F(node_x) = i$ if $node_x$ processes the i^{th} affair, or acts as the ith role.

Another is based on the dependency relations or dependency function R defined above. If two nodes lie in the same category, these two nodes have the same dependency relations with any other nodes in the system. If their dependency relations with other nodes are different, these two nodes belong to different categories. This rule is described as follows.

$$\begin{aligned} &\forall node_x,\ node_y \in S \\ &if\ \exists k,\ and\ node_x \in U_k, node_y \in U_k \\ &then\ \forall node_i \in S \\ &R(node_i, node_x) = R(node_i, node_y), \quad and \\ &R(node_x, node_i) = R(node_y, node_i) \end{aligned} \tag{6}$$

Comments are also needed here. This rule is not a necessary and sufficient condition, e.g., two nodes in different categories do not mean they have different dependency relations with other nodes in the system.

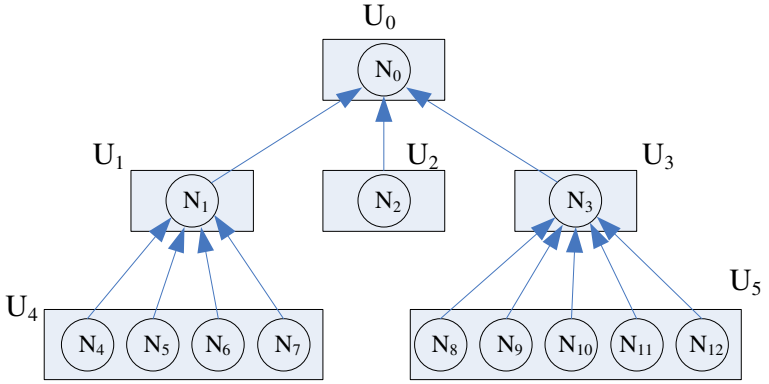

Fig. 1 Nodes dependency between nodes in cluster computing systems

While combining the above two rules together, we can get the necessary and sufficient condition of two nodes $node_x$, $node_y$ belonging to the same category U_k.

$$\begin{aligned}
&\forall node_x, node_y \in S, if\ \exists k, and, node_x \in U_k, node_y \in U_k \\
&\Leftrightarrow \\
&F(node_x) = F(node_y), \\
&and,\ \exists node_i \in S, \\
&R(node_i, node_x) = R(node_i, node_y), \\
&R(node_x, node_i) = R(node_y, node_i)
\end{aligned} \tag{7}$$

where $\Leftrightarrow$ means the necessary and sufficient condition.

Let us give an example to illustrate the classification of nodes in a cluster computing system. Fig. 1 is a cluster computing system S with 13 nodes, named $N_0, N_1, \ldots, N_{12}$. These nodes are divided into 6 categories, named $U_0, U_1, \ldots, U_5$. $N_i \rightarrow N_j$ indicates node N_i depends on node N_j, that is $R(N_i, N_j) > 0$. Based on the above classification rules, we can divide 13 nodes into 6 categories, and nodes in each box belong to the same category. Function values $F(N_i)$ of nodes in the same category U are equal, but are different in different categories. In effect, the classification of nodes in a cluster computing system is based on both function $F(N_i)$ and function $R(N_i, N_j)$.

3.2 The SRN model of cluster computing systems

Using Stochastic Reward Nets (SRN), cluster systems which employ software rejuvenation were modeled and analyzed in [11,12,26,29]. Liu et.al. [26] developed SNR models to evaluate different software rejuvenation policies deployed in a Cable Modem Termination System cluster system. The numerical results from the proposed models showed that the significant availability improvement and downtime cost reduction when either the time-based or measurement-based software rejuvenation strategies were introduced in the cluster system. Vaidyanathan et.al. [12] modeled and analyzed

cluster systems which employed software rejuvenation using SRN models. They then briefly described implementation of software rejuvenation that performed periodic and predictive rejuvenation operations, and showed some empirical data from real life systems that exhibit software aging. Wang et.al. [29] analyzed fixed and delayed software rejuvenation policies of cluster server systems under varying workload. SNR models were constructed for a WWW cluster server system with above two two software rejuvenation policies. Fan et.al. [11] modeled the dispatch–worker based cluster systems, in which the dispatcher received service requests from clients and selected a worker from the worker pool to process the requests. They proposed a SNR model for dispatcher–worker based architecture with prediction-based rejuvenation. Compared with cluster systems of a flat architecture, in which all the nodes share the same functions, dispatcher–worker based cluster systems employed different software rejuvenations on the dispatcher and the worker pool, respectively. However, one of the limitations of these SRN models was that they did not consider dependent relation between nodes in a cluster system, and failed to study the effects of dependency between nodes on the whole cluster system. As a complement, this chapter takes into account the dependency between nodes in a cluster system. Furthermore, we model dependency between nodes explicitly, and analyze the effect of dependency on the whole cluster system. Based on the discussion above, we reconstruct a SRN model. The whole system model is composed of many interrelated sub-models. Each sub-model corresponds to a category. Nodes in the same category constitute a sub-model, and all these sub-models form the complete model of the whole cluster a computing system. Different sub-models have the same places, transitions and directed arcs, but the tokens in the places and the weights to the arcs are different.

Figure 2 describes the SRN sub-model of a category U_i. In this study, we use the following symbols to characterize this sub-model:

(a) the circles represent the places, the open rectangles denote transitions with exponentially distributed firing time while the shaded rectangles indicate transitions with a constant firing time.
(b) i indicates that this sub-model corresponds to the ith category.
(c) place S_{i_0} models the robust and healthy state soon after the node i is started.
(d) the exponentially distributed transition T_{i_fprob} models the first stage of software aging, i.e., the software enters failure probable state S_{i_fprob}.
(e) S_{i_fprob} indicates that the node is in unstable state, in which the node is still operational but its performance degrades as time progresses. The nodes are still operational in this state but can fail (transit to place S_{i_fail} with a non-zero probability).
(f) place S_{i_fail} indicates that the node is in the failure state. A node in this state can be repaired with probability α through the transition $T_{i\_noderepair}$, and it can fail with probability $1-\alpha$, leading to a system failure. Thus the time to failure for the node starting in the robust state S_{i_0} has an hypo-exponential distribution. Since this is an increasing failure rate distribution, it models software aging.
(g) transition $T_{i\_nodefail}$ models the second stage of software aging, the firing of which will cause software failure, modeled by S_{i_fail}.
(h) transition $T_{i\_noderepair}$ denotes that the node enters the robust state after recovery.

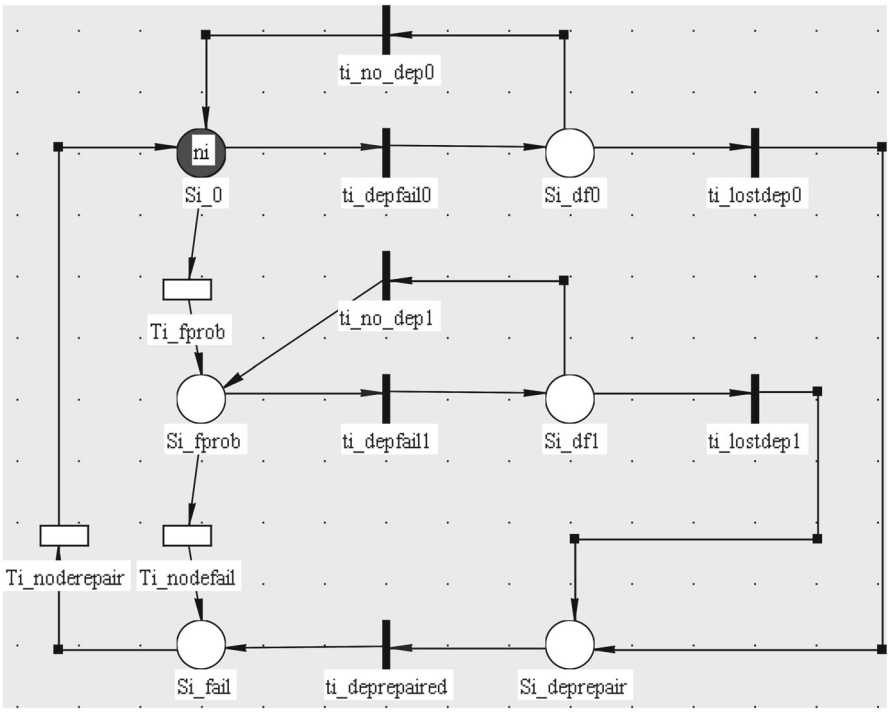

Fig. 2 The SRN sub-model of a category U_i

(i) place S_{i_df0} models that the depended node fails and the depending nodes which are in the normal state enter this state.
(j) $t_{i\_depfail0}$ is the transition from place S_{i_0} to place S_{i_df0}. Nodes in S_{i_df0} have probability c_i to work with depended node, and nodes in S_{i_df0} enter into place $S_{i\_deprepair}$ with transition $t_{i\_depfail0}$. With probability 1-c_i (transition $t_{i\_no\_dep0}$), the node will work independently.
(k) the place $S_{i\_deprepair}$ denotes that depending nodes keep waiting till the failure depended node has been repaired.
(l) after the depended node is repaired, the depending nodes enter into place S_{i_fail} through transition $t_{i\_deprepaired}$.
(m) nodes in S_{i_df0} have probability $1-c_i$ to work by themselves, and nodes in S_{i_df0} enter into place S_{i_0} through transition $t_{i\_no\_dep0}$.
(n) the place S_{i_df1} has the similar means with the place S_{i_df0}, and transition $t_{i\_depfail1}$ also has the similar means with transition $t_{i\_depfail0}$.
(o) the transition $t_{i\_no\_dep1}$ has the similar means with transition $t_{i\_no\_dep0}$, and the transition $t_{i\_lostdep1}$ also has the similar means with transition $t_{i\_lostdep0}$ except that tokens comes from S_{i_fprob} not from S_{i_0}.

Repairing actions are carried out on the failure nodes, and these repaired nodes then return to the normal state. The sub-model in Fig. 2 describes the ith category U_i. At the beginning, the system is in normal state indicated by n_i tokens in S_{i_0}. Here n_i

is nodes number contained in U_i. All nodes are in the normal state when the cluster computing system starts. With the growth of the running time, there is one or more tokens (i.e., nodes) enter into the unstable state S_{i_fprob} through transition T_{i_fprob}. In unstable state S_{i_fprob}, nodes can still work, but the performance has declined. Then tokens enter into failure state S_{i_fail} from unstable state S_{i_fprob} through transition $T_{i\_nodefail}$. In the cluster computing system, robust nodes usually are not regarded entering into failure state only through a transition. Before nodes fail, they enter into an unstable state first. In unstable state, nodes still work, but they are under surveillance. For those nodes in the failure state, they will continue to work after they are repaired properly. In this model, it is expressed that tokens in place S_{i_fail} come back to place S_{i_0} through transition $T_{i\_noderepair}$.

Nodes in working state include those nodes in the normal state S_{i_0} and in the unstable state S_{i_fprob}. If the depended node which is in the working state fails suddenly, the depending nodes in normal state (i.e., nodes in place S_{i_0}) enter into place S_{i_df0} through transition $t_{i\_depfail0}$, and nodes in the unstable state (i.e., nodes in place S_{i_fprob}) enter into place S_{i_df1} through transition $t_{i\_depfail1}$. We consider that the failure of the depended node affects the depending working nodes with probability c_i. For example, the depending working nodes spend c_i of their working time to get data from the depended node, and spend $1 - c_i$ of their working time to work by themselves. In this model, it is expressed that tokens in place S_{i_df0} enter into place $S_{i\_deprepair}$ through transition $t_{i\_lostdep0}$ with probability c_i, and tokens in place S_{i_df1} enter into place $S_{i\_deprepair}$ through transition $t_{i\_lostdep1}$ with probability c_i. It is also noted that tokens in place S_{i_df0} and S_{i_df1} also have $1 - c_i$ probability come back to S_{i_0} and S_{i_fprob}, respectively. Nodes entering into place $S_{i\_deprepair}$ must wait till its depended node is repaired through transition $t_{i\_deprepaired}$. Then nodes in place $S_{i\_deprepair}$ enter into place S_{i_fail}. Nodes in place S_{i_fail} then go back to place S_{i_0} through transition $T_{i\_noderepair}$ to start their normal working.

3.3 The SRN model of software rejuvenation in cluster computing systems

In order to reduce the loss of system failure, we introduce software rejuvenation policy into our proposed SRN model. To avoid the unexpected shutdown of the system, some rejuvenation actions are carried out according to certain policies. We add some places and transitions in the model shown in Fig. 2, and obtain the new model described in Fig. 3. The place $S_{i\_deprejuv}$ denotes that the depending nodes enter into this state to wait when the depended node is carried out rejuvenation actions. After the depended node completes its rejuvenation actions, the depending nodes enter into place S_{i_rejuv} to carry out rejuvenation actions. After the rejuvenation actions of the depending nodes are finished, the system comes back to normal state S_{i_0}, and a new cycle starts. The transition from the place S_{i_rejuv} to normal state S_{i_0} is given by T_{i_rejuv}.

Place S_{i_rejuv} denotes nodes carry out rejuvenation actions. When nodes are in this state, they are unavailable. Because the rejuvenation action is proactive and preventive, the loss of transitory downtime is lower than the loss caused by failure. Without rejuvenation, nodes eventually enter into the failure state S_{i_fail} through transition $t_{i\_norejpolicy}$, place $S_{i\_nopolicy}$, and transition $T_{i\_nodefail}$. Before nodes enter into state

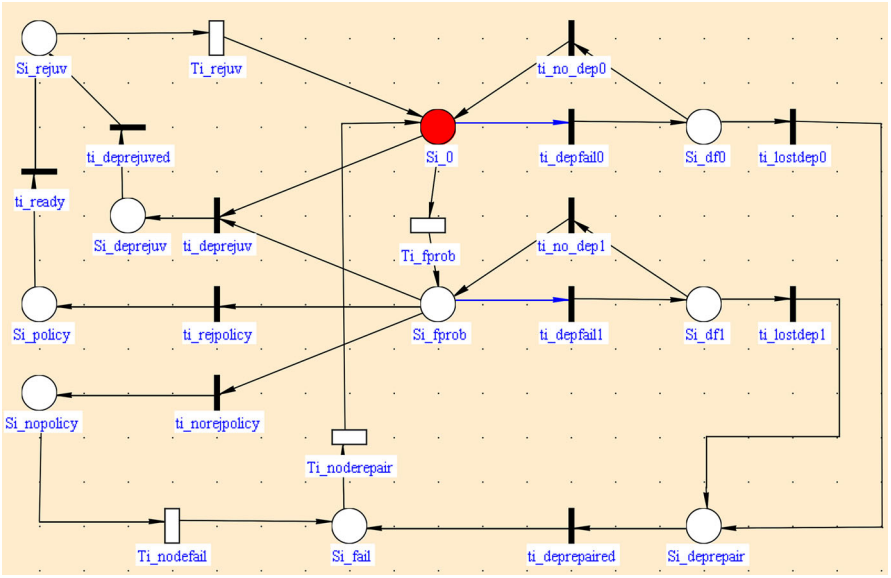

Fig. 3 The SRN model of software rejuvenation

S_{i_fail}, if they enter into state S_{i_policy} through transition $t_{i\_rejpolicy}$, then enter into state S_{i_rejuv} through transition t_{i_ready}, the rejuvenation actions are carried out. For those nodes are in unstable state (i.e., nodes in place S_{i_fprob}), we can carry out software rejuvenation operations in advance before they fail. For different types of nodes, we can use different software rejuvenation strategies. For example, computational nodes can be used timing software rejuvenation strategy. Timing software rejuvenation strategy is appropriate for those nodes that restoration cost is not high. Resource-dependent nodes can be used predicting software rejuvenation strategy. Predicting software rejuvenation strategy is applicable to those nodes that their resources have great impact on their performance.

As for those central processing nodes, we may use a more sophisticated software rejuvenation strategy. Integrated timing and predict strategies are suitable for those nodes where their repair costs are high. In the SRN model, we can carry out different software rejuvenation strategies through selecting different values of probability parameter p_i. When nodes are in unstable state, i.e., nodes in place S_{i_fprob}, they perform software rejuvenation operations with probability p_i. Tokens in place S_{i_fprob} reach place S_{i_rejuv} via transition $t_{i\_rejpolicy}$, place S_{i_policy} , and transition t_{i_ready}, then return to normal state (i.e., S_{i_0}) through transition T_{i_rejuv}. In addition, nodes in place S_{i_fprob} don't carried out software rejuvenation operations with probability $1 - p_i$. In this case, tokens in place S_{i_fprob} reach place $S_{i\_nopolicy}$ through transition $t_{i\_norejuvpolicy}$, and reach place S_{i_fail} via transition $T_{i\_nodefail}$, which means that nodes fail. After nodes fail, they can go back to normal state (i.e., S_{i_0}) through transition $T_{i\_noderepair}$.

After cluster computing system is running, most nodes work normally, which means that most tokens are in place S_{i_0} and S_{i_fprob}. When a depended node starts software

rejuvenation operation (i.e., the token of the depended node is in place S_{i_rejuv}), token of depending node enters into place $S_{i\_deprejuv}$ from place S_{i_0} and S_{i_fprob} through transition $t_{i\_depjejuv}$. After the depended node completes its software rejuvenation operation (i.e., token of the depended node is returned to place S_{i_0}), depending node starts its software rejuvenation operation (i.e., tokens of depending node enter place S_{i_rejuv} from place $S_{i\_deprejuv}$ through transition $t_{i\_deprejuved}$). It can be detected whether a depended node is carrying out software rejuvenation operation or not through enabling function of transition $T_{i\_deprejuv}$. Also it can be detected whether a depended node completes its software rejuvenation operation or not through enabling function of transition $T_{i\_deprejuved}$. The enabling function of transition $T_{i\_deprejuv}$ is defined as follows.

$$e_{i\_deprejuv} = \sum_{k=0}^{n-1} (\#S_{rejuv}(U_p(node_k)))R(node_i, node_k) > 0 \tag{8}$$

In Eq. (8), $U_p(node_k)$ denotes the specific category U_p which the $node_k$ belongs to. And $\#S_{rejuv}(U_p)$ represents the number of tokens which are in place S_{i_rejuv} in the SRN sub-model of category U_p. By using this enabling function in Eq. (8), we can obtained the depended node $node_k$. The enabling function of transition $T_{i\_deprejuved}$ is defined as follows.

$$e_{i\_deprejuved} = not(e_{i\_deprejuv}) \tag{9}$$

In Eq. (9), the enabling function of $e_{i\_deprejuved}$ is the negative of function $e_{i\_deprejuv}$. Obviously, both enabling functions of $e_{i\_deprejuved}$ and $e_{i\_deprejuv}$ are boolean functions. Enabling function is a new added function in SRN, and in the conditional Petri net, there is not such a advanced function. When the conditions in the enabling function are satisfied, all the tokens in the place can be transferred to the specific place through the particular transition. Thus, we only need to check if there are failure nodes in the SRN sub-model of category U_p which the $node_k$ belongs to.

Comments are added on the selection of parameter p_i. For those nodes where the loss of failure is small, or failing period is long from the normal state to the failure state, we can choose a small value for p_i in order to prevent frequent software rejuvenation operations. On the contrary, for those nodes where the loss of failure is large, or failing period is small from the normal state to the failure state, we can choose a large value for p_i in order to have enough time to perform software rejuvenation operations. Another comment is on the dependency relation between nodes in our model. When a node is at work (i.e., nodes in the normal state S_{i_0}, or in an unstable state S_{i_fprob}), the depended node may have to carry out software rejuvenation operations. At this time, the depending nodes have to wait till the depended node completes its software rejuvenation operation, then the depending nodes perform their software rejuvenation operations. The reason to need thus operational requirement is that during the period when software rejuvenation operation is performed, nodes are not available. When the depended node carries out software rejuvenation operation, it affects on the depending nodes' working. In effect, when the depended node carries out software rejuvenation operation, the depending nodes enter into place S_{i_df0} and place S_{i_df1},

further enter into place S_{i_fail} with a certain probability. In order to prevent depending nodes fail when the depended node performs software rejuvenation operation, we enforce depending nodes to carry out software rejuvenation after the depended node completes its software rejuvenation operation. Although this design increases the cost of depending nodes' software rejuvenation operations, we can avoid depending nodes failing effectively. The next simulation results also prove the effectiveness of this design.

4 Simulation experiments and evaluations

This section outlines the results from experiments. Specifically, we simulate the SRN model of cluster computing system proposed in this paper, and discuss the effect of the model on the availability of the cluster system. The SRN model shown in Fig. 3 is solved using the stochastic petri net software package (SPNP) which can automatically generate the reachable graph and solve the model numerically for both transient and steady state analysis.

4.1 Relations among t_{fprob}, p_i and the system cost

In this test case, we analyze the relations among t_{fprob}, p_i and the system cost. It is considered that $\forall node_x,\ node_y \in S, R(node_x, node_y) = 0$. In other words, we ignore the dependency relation between nodes, all nodes are in one category. The objective is to analyze the impact of various parameters on the cluster computing system on the premise of not considering dependency relation between nodes. We also consider that the time duration of a node entering into the unstable state from the normal state is t_{fprob}, and the time duration entering into the failure state from the unstable state is $t_{nodefail}$. However, in a real cluster computing system, we are only able to identify the time duration of a node from the beginning of the normal state to failure state. We don't know the exact moment when a node enter into the unstable state. That is, we only know the $t_{worktime}$ as follows.

$$t_{worktime} = t_{fprob} + t_{nodefail} \tag{10}$$

First we analyze the relation between t_{fprob} and the probability p_{rejuv} of tokens entering into place S_{rejuv}. We consider that the cluster computing system only has one node, and set parameter $t_{worktime} = t_{fprob} + t_{nodefail} = 1000.0,\ t_{noderepair} = 50.0,\ t_{rejuv} = 10.0,\ p_i = 0.8$. Test data is shown in Table 1.

Let the time t_{fprob} of a node entering into the unstable state from the normal state as x axis, and probability p_{rejuv} of tokens entering into place S_{rejuv} as y axis. In Fig. 4, we can find that in other conditions remain unchanged probability p_{rejuv} of tokens entering into place S_{rejuv} decreases when t_{fprob} increases.

The relation between t_{fprob}, p_i and the system cost is analyzed in Fig. 5. It is considered that the cost that system works normally is C_0, the cost that system fails is C_{fail}. In general, C_{fail} is much larger than C_0. The average system cost should be the sum of probability tokens entering into places multiplied by the fixing costs of the

Table 1 Relations between t_{fprob} and p_{rejuv}

t_{fprob}	$t_{nodefail}$	$p_{rejuv}(\times 0.01)$
100.0	900.0	2.685
200.0	800.0	2.116
300.0	700.0	1.747
400.0	600.0	1.487
500.0	500.0	1.295
600.0	400.0	1.146
700.0	300.0	1.028
800.0	200.0	0.932
900.0	100.0	0.853

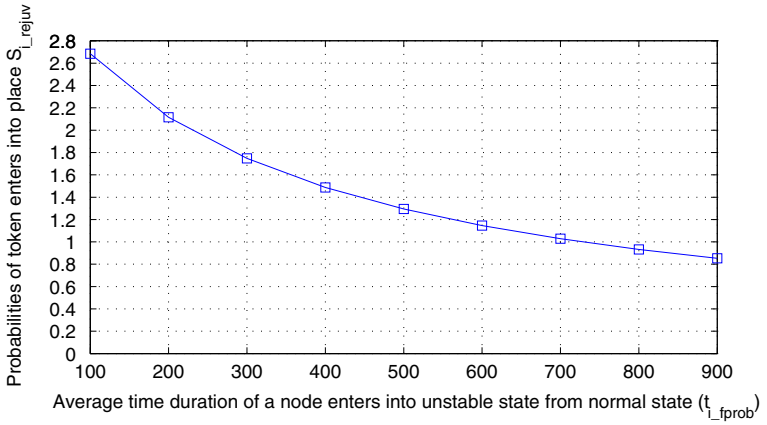

Fig. 4 Relations between t_{fprob} and probability p_{rejuv} of tokens entering into place S_{rejuv}

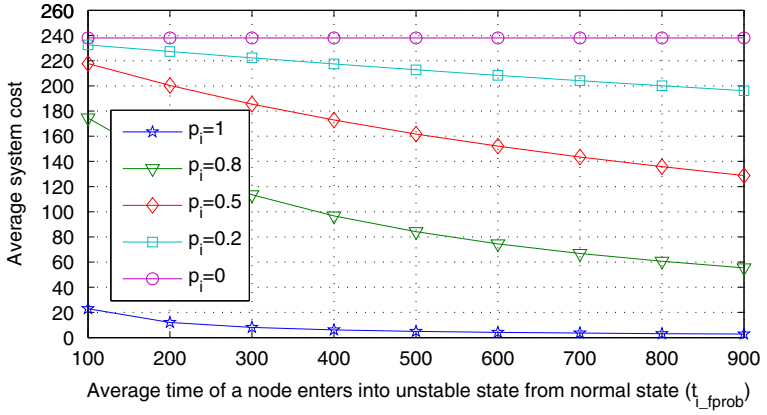

Fig. 5 Relations between t_{fprob}, p_i and the system cost

Table 2 Relations among t_{fprob}, p_i and the system cost

t_{fprob}	$p_i = 1$	$p_i = 0.8$	$p_i = 0.5$	$p_i = 0.2$	$p_i = 0$
100.0	22.93	174.5	217.7	232.6	238.1
200.0	12.02	137.6	200.4	227.3	238.1
300.0	8.142	113.5	185.6	222.3	238.1
400.0	6.160	96.65	172.9	217.5	238.1
500.0	4.954	84.14	161.8	212.8	238.1
600.0	4.148	74.50	152.1	208.4	238.1
700.0	3.566	66.84	143.5	204.2	238.1
800.0	3.124	60.61	135.8	200.1	238.1
900.0	2.770	55.44	128.8	196.2	238.1

places. So if we can reduce the probability of tokens entering into places with high fixing cost, we can reduce the total cost of the whole system. Software rejuvenation is a policy to reduce the probability of tokens entering into failure states where fixing cost is very large.

Let C_{i_0} denotes the average cost of place S_{i_0}, C_{i_fprob} represents the average cost of place S_{i_fprob}, C_{i_df0} indicates the average cost of place S_{i_df0}, C_{i_df1} means the average cost of place S_{i_df1}, $C_{i\_0} = C_{i\_fprob} = C_{i\_df0} = C_{i\_df1}$, $C_{i\_deprepair}$ denotes the average cost of place $S_{i\_deprepair}$, C_{i_fail} is the average cost of place S_{i_fail}, $C_{i\_deprepair} = C_{i\_fail}$, C_{i_rejuv} denotes the average cost of place S_{i_rejuv}, $C_{i\_deprejuved}$ indicates the average cost of place $S_{i\_deprejuved}$, C_{i_policy} represents the average cost of place S_{i_policy}, $C_{i\_nopolicy}$ means the average cost of place $S_{i\_nopolicy}$, $C_{i\_rejuv} = C_{i\_deprejuved} = C_{i\_nopolicy} = C_{i\_policy}$. Obviously, we should reduce the probability of tokens entering into places $S_{i\_deprepair}$ and S_{i_fail}. Let $C_{i\_0} = 0$, $C_{i\_fail} = 5{,}000$, and $C_{i\_rejuv} = 250$. Test data is as shown in Table 2.

Let the time t_{fprob} that a node enters into the unstable state from the normal state as x axis, and the system cost as y axis. Figure 5 shows the relations among t_{fprob}, p_i and the system cost.

In our proposed model, parameter p_i means the probability of successful detection of software rejuvenation. $p_i = 1$ means completely successful detection of software rejuvenation. $p_i = 0$ means cancel of the rejuvenation strategy. From Fig. 5 we can see that the system's average cost decreases with the increasing of p_i. In this model, parameter t_{fprob} controls the frequency of rejuvenation operation. In simulations, nodes fail after 1,000 time units. The ideal time to execute rejuvenation operation is close to 1,000 (for example, in 900). If rejuvenation operation is performed too early, for instance in 20, rejuvenation undoubtedly increase the cost of system, and adds unnecessary overhead to the system. Based on the above test data, we can conclude: (1). The more accurate rejuvenation forecast (i.e., p_i is close to 1), the smaller the average cost of the system; (2). When the rejuvenation operation interval is closer to the working time $t_{worktime}$ of the system, the smaller the average cost of the system is. (3). The optimal p_i will be found. If the time t_{fprob} of a node is known, we can set p_i=1. In this situation, we set the interval of rejuvenation operation slightly less than time t_{fprob}, and can obtain the optimal solution. In a real system, the time t_{fprob} of a node is unknown. It is hard to set p_i to an optimal value because rejuvenation

Table 3 Relations among t_{fprob}, p_i, and system averagecost

t_{fprob}	p_i	Average cost
100.0	0.1	235.62
200.0	0.2	227.32
300.0	0.3	212.26
400.0	0.4	190.18
500.0	0.5	161.86
600.0	0.6	129.13
700.0	0.5	143.47
800.0	0.4	158.28
900.0	0.3	174.36

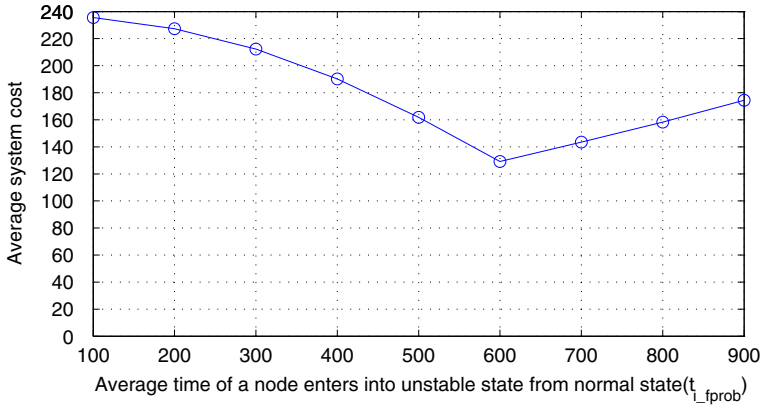

Fig. 6 The optimal value of p_i

detection module needs to dynamically configure p_i according to the judgments on the nodes working situations. In this model, selection of p_i is associated with t_{fprob} and $t_{nodefail}$. When t_{fprob} is small, system is in a relatively stable working state, and there is not need to do rejuvenation operation. In this case, the value of p_i is also small. When t_{fprob} increases, the probability of the node failure also increases. In this case, the value of p_i need increase. The test data describe the above relations among t_{fprob}, p_i, and the system average cost are shown in Table 3.

We use the time t_{fprob} that a node enters into the unstable state from the normal state as x axis, and the system cost as y axis. Figure 6 shows the optimal value of p_i. From Fig. 6, we can see that the relation between rejuvenation interval and cost of the system is not a simple linear relation. Through the simulation data, we can calculate the optimal value of p_i. In that case, the system average cost is minimized.

Figure 6 shows the similar relation between system average cost and rejuvenation interval with that in [26,12]. If the rejuvenation interval is very small, the system is always rejuvenation and thus incurs high cost and downtime. As the rejuvenation interval increases, both expected unavailability and cost incurred decrease and reach an optimum value. If the rejuvenation interval goes beyond the optimal value (i.e., we rejuvenate less and less frequently), the system failure has more influence on these measures than rejuvenation. Hence the average cost begins to increase. In Fig. 6, the

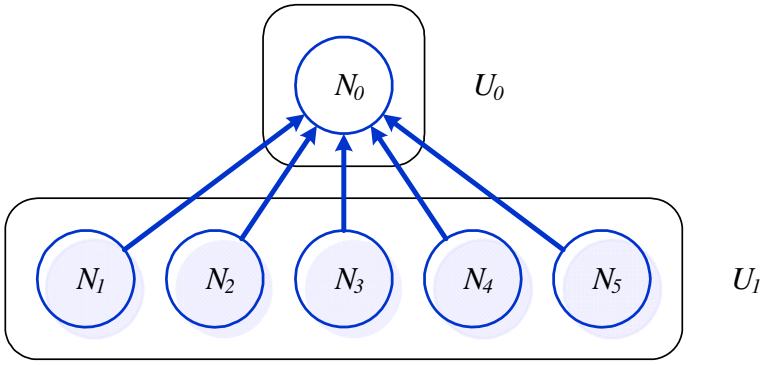

Fig. 7 A cluster computing system with dependency relations between nodes

Table 4 Parameters

Parameter name	U_0	U_1
n	1	5
t_{fprob}	4, 000	600
$t_{nodefail}$	1, 000	400
$t_{noderepair}$	100	50
t_{rejuv}	20	10
C_{fail}	20, 000	5, 000
C_{rejuv}	300	200

minimum for the system cost occurs at 600, which is larger than the half point of the $t_{worktime}$. It is consistent with the modeled software aging processes in Figs. 2 and 3. Because we use a hypo-exponential distribution to model the software aging in our SRN models in Figs 2 and 3, in which the exponentially distributed transition T_{i_fprob} the first stage of software aging, i.e., the software enters failure probable state S_{i_fprob}, and transition $T_{i\_nodefail}$ models the second stage of software aging, the firing of which will cause software failure. Hypo-exponential distribution is an increasing failure rate distribution, and t_{fprob} is usually longer than $t_{nodefail}$. When a node enters into space S_{i_fprob} in which the node is in unstable state, the optimal time for rejuvenation occurs.

4.2 With dependency relations between nodes in the model

In this test case, we analyze the cluster computing system which has dependency relation between nodes. We assume the cluster computing system S are as shown in Fig. 7.

N_0 is the central node (i.e., N_0 is the depended node), and nodes $N_1, N_2, \ldots, N_5$ depend on central node N_0(i.e., $N_1, N_2, \ldots, N_5$ are depending nodes). Based on the classification rules proposed in Sect. 2, we have $U_0 = \{N_0\}, U_1 = \{N_1, N_2, N_3, N_3, N_5\}$. We set $R(U_1, U_0) = 1$. Other parameters are set as shown in Table 4. To compare the differences of results before and after the introduction of

Table 5 No software rejuvenation operation

Parameter name	Without dependency	With dependency
p_{0_fail}	0.01965	0.01965
p_{1_fail}	0.07912	0.1132
$p_{1\_depfail}$	0.0000	0.01896
Cost	1,212	2,071

Table 6 With software rejuvenation

Parameter name	Without dependency	With dependency
p_{0_fail}	0.000002487	0.000002487
p_{1_fail}	0.000008333	0.00001449
$p_{1\_depfail}$	0.0000	0.000002487
p_{0_rejuv}	0.004974	0.004974
p_{1_rejuv}	0.01667	0.02891
$p_{1\_deprejuv}$	0.00	0.004970
Cost	4.973	12.62

the dependency relation between nodes, we get two group of test data. One is obtained with dependency relation between nodes, another is obtained without dependency relation between nodes. For convenience, we temporarily close rejuvenation strategy (i.e., we set $p_0 = 0,\ p_1 = 0$). The test results are shown in Table 5.

We can observe that the dependency relations between nodes affect the probability of tokens entering into some places and the system average cost. Without a dependency relation between nodes (i.e., $\forall i,\ j,\ R(N_i, N_j) = 0$), it is just as two separate cluster computing systems. Each system has its own properties, and there is no interference between them. When we simulate the two separate systems, respectively, we can get the same results. If we set up the value of $R(N_i, N_j) = 1$ and then simulate again, we obtain the different results. Node N_0 is the center node. If node N_0 fails, all the depending nodes, $N_1, N_2, \ldots, N_5$, will not work normally. Comparing the two group results in Table 5, we can see that the failure probability p_{0_fail} of center node N_0 does not change before and after the introduction of dependency relations. But the failure probability p_{1_fail} of depending nodes $N_1, N_2, \ldots, N_5$ increases by 0.0341 after the introduction of dependency relations. Obviously, the increased probability value is caused by the failure of the depended node N_0. Increasing in the average cost of system is caused due to the increasing in p_{1_fail}. From the above test and analysis, we can see that model with dependency relations between nodes can describe the cluster computing system characteristics better.

4.3 Software rejuvenation analysis

In this test case we add rejuvenation strategy for the system. We set p_0=0.9, p_1=0.9, other parameters are not changed. The test results are shown in Table 6.

From Table 6, we can see that the average cost of system reduces greatly after adding software rejuvenation strategy. In considering the dependency relations between nodes,

the failure probability and rejuvenation probability of nodes in U_1 increase. And the average cost increases to three times as that of the system without dependency relations. From these test data, we can analyze the impact more clearly of dependency relations between nodes in cluster computing system on software rejuvenation. Without considering the dependency relation between nodes, we can adjust parameters for each category nodes, so that the average cost of the system is minimum. But, while considering the dependency relations, it is needed to consider the relations betweens nodes in the entire system to set the parameters in order to make the system average cost to a local optimum value.

5 Implementation of software rejuvenation in Smart Platform

In this section, we present the implementation of software rejuvenation in Smart Platform [30], which is a software supporting platform in smart interactive space. The main role of Smart Platform is to connect software modules distributed in a number of intelligent interactive space and to provide them with the underlying software support and the corresponding software tools in order to achieve the collaboration between them so as to meet the requirements of a distributed computing environment in smart interactive space. Figure 8 shows the Smart Platform composition of the elements and their position in smart space. Smart Platform runs on multiple interconnected computers distributed in intelligent interaction space. It provides a unified running environment for software modules which run on it. Each software module is encapsulated into an Agent. In each computer, there runs a module Container as the container for each Agent. There is a central server, Directory Service, in the Smart Platform computing environment. Directory Service provides operations, such as name querying, information forwarding, debugging, monitoring and other services, for all Agents. Container

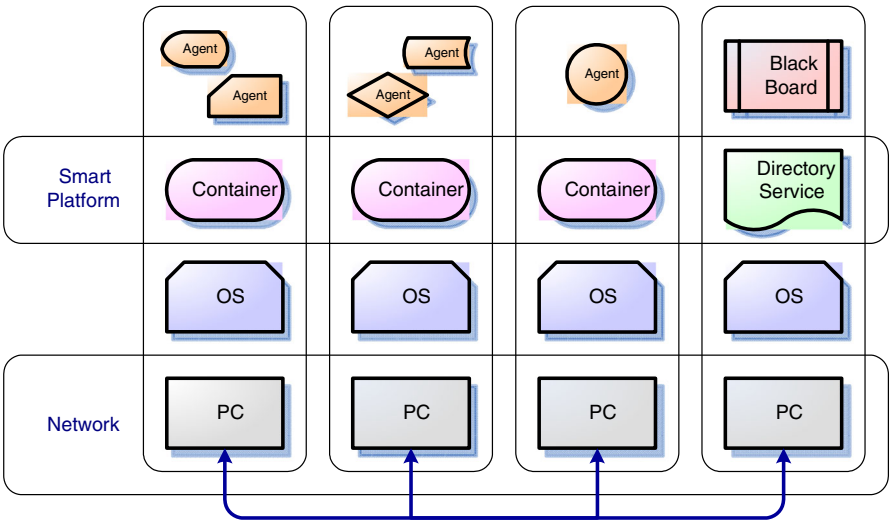

Fig. 8 Functional architecture of Smart Platform

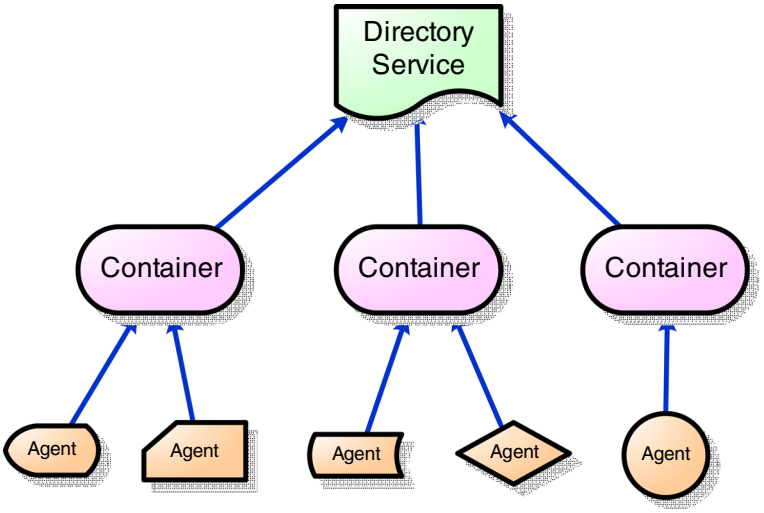

Fig. 9 Dependency relations between nodes in Smart Platform

is responsible for connecting Agent and Directory Service. It shields the underlying communication details for Agents, and provides a unified communication interface for all Agents. Smart Platform is a typical distributed cluster computing system with the dependency relation between nodes. It is consistent with the cluster computing system model proposed in Sects. 3 and 4. The dependency relations among Directory Service, Container and Agent are shown in Fig. 9. Directory Service is the center node, and Container depends on Directory Service. Agent depends on Container. Container failure causes Agent to stop working, while the Directory Service failure will result in Container failure. We develop software rejuvenation strategy for these three modules respectively. Smart Platform supports the adoption of self-describing XML file to read the Agent information, so we write the software rejuvenation strategy in the self-describing XML file. In order to facilitate users to configure software rejuvenation strategy for Container and Directory Service, we have to add a configuration file Config.xml. After users write the rejuvenation strategy in the Config.xml file, Container and Directory Service automatically read their contents while they run.

In this paper, we only test the Agent performance. However, while performing the same process of testing on Directory Service and the Container, we can obtain corresponding results of Directory Service and Container. Test environment includes two network connected computers, a computer is equipped with a Directory Service, another with a Container and an Agent. To shorten the test time, we let Agent not release the allocated memory deliberately. We record memory usage regularly to check the performing of software rejuvenation operation. Memory usages in Agent are shown in Fig. 10. From Fig. 10, we can see that, in the case of absence of software rejuvenation, memory usages in the Agent continue to increase with the Agent running time increasing. After some time, the Agent uses up the physical memory of the computer. Ultimately, only two possibilities take place, one is that requests to allocate memory from Agent is rejected and the Agent fails, another is that the Agent will

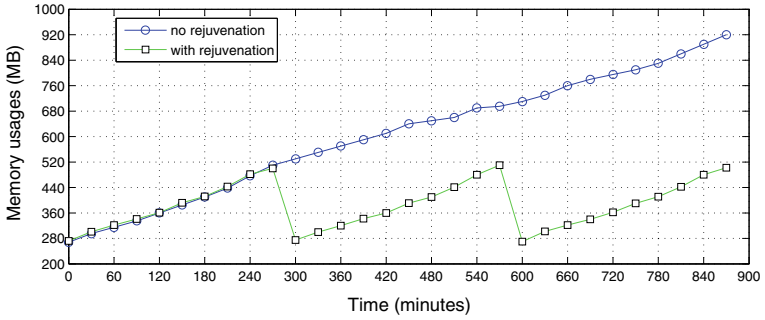

Fig. 10 Memory usages in the Agent

lead to the operating system crash. After introducing software rejuvenation operation, monitoring module regularly checks the memory usage of Agent. Once it finds that the memory usage of Agent exceeds the user specified threshold, monitoring module shuts down the Agent process, and then restarts it. From Fig. 10, we can see that the memory usage of Agent fluctuates within user specified thresholds. The implementation method of software rejuvenation in the Smart Platform can also be applied to other cluster computing systems.

6 Conclusions

In this paper, we included the dependency between nodes into cluster systems, and investigated software rejuvenation for this cluster systems. We were able to clearly demonstrate that the dependency between nodes has effect on software rejuvenation policy which applied in this cluster systems. Specifically, we first have investigated cluster computing systems with dependency between nodes. Then we reconstructed an SRN model for the software rejuvenation in the cluster computing system which has dependency between nodes. After simulation on the SRN model, we can draw the following conclusions: (1) The more accurate rejuvenation forecast (i.e., p_i is close to 1), the smaller the average cost of the system. (2) When the rejuvenation operation interval is closer to the working time $t_{worktime}$ of the system, the smaller the average cost of the system is. (3) The relation between rejuvenation interval and the cost of the system is not a simple linear relation. (4) While considering the dependency between nodes, it is needed to consider the relations betweens nodes in the entire system to set the parameters in order to make the system average cost reach a local optimum. Finally, based on the analysis, a prototype of software rejuvenation in Smart Platform has been developed. After implementation and analysis of software rejuvenation policy in Smart Platform, we not only describe the principle and behavior of the software rejuvenation, but also propose rejuvenation implementation method in a real life cluster computing system.

References

1. Parnas D (1994) Software aging. In: Proceedings of the 16th international conference on software engineering, pp 279–287

2. Huang Y, Kintala C, Kolettis N, Fulton N (1995) Software rejuvenation: analysis, module and applications. In: Proceedings of 25th symposium on fault tolerant, computing, pp 381–390
3. Grottke M, Li L, Vaidyanathan K, Trivedi K (2006) Analysis of software aging in a web server. IEEE Trans Reliab 55(3):411–420
4. Matias R, Freitas P, (2006) An emperimental study on software aging and rejuevenation in web servers. In: Proceedings of 30th annual international conference on computer software and applications, vol 1, pp 189–196
5. Grottke M, Nikora A, Trivedi K (2010) An empirical investigation of fault types in space mission system software. In: Proceedings of IEEE conference on dependable systems and networks, pp 447–456
6. Moorsel A, Wolter K (2006) Analysis of restart mechanisms in software systems. IEEE Trans Softw Eng 32(8):547–558
7. Alonso J, Torres J, Berral J, Gavalda R (2010) Adaptive on-line software aging prediction based on machine learning. In: Proceedings of international conference on dependable systems and networks, pp 507–516
8. Dugan J, Trivedi K (1989) Coverage modeling for dependability analysis of fault-tolerant systems. IEEE Trans Comput 38(6):775–787
9. Gokhale S, Trivedi K (1998) Dependency characterization in path-based approaches to architecture-based software reliability prediction. In: Proceedings of international conference on application-specific software engineering technology, pp 86–89
10. Popstojanova K, Trivdei K (2000) Failure correlation in software reliability models. IEEE Trans Reliab 49(1):37–48
11. Fan X, Xu G, Ying R, Zhang H, Jiang L (2003) Performance analysis of software rejuvenation on dispatcher–worker based cluster system. In: Proceedings of the 4th international conference on parallel and distributed computing, applications and technologies, pp 562–566
12. Vaidyanathan K, Haarper R, Hunter S, Trivedi K (2001) Analysis and implementation of software rejuvenation in cluster systems. In: Proceedings of joint international conference on measurement and modeling of computer systems, ACM SIGMETRICS, pp 62–71
13. Bobbio A, Sereno A, Anglano C (2001) Fine grained software degradation models for optimal rejuvenation policies. J Perform Eval 46:45–62
14. Dohi T, Popstojanova K, Trivedi K (2000) Statistical nonparametric algorithms to estimate the optimal software rejuvenation schedule. In: Proceedings of Pacific rim international symposium dependendable computing, pp 77–84
15. Grag S, Puliafito A, Telek M, Trivedi K (1998) Analysis of preventive maintenance in transactions based software systems. IEEE Trans Comput 47(1):96–107
16. Bao Y, Sun X, Trivedi K (2005) A workload-based analysis of software aging and rejuvenation. IEEE Trans Reliab 55(3):541–548
17. Koutras V, Platis A, Gravvanis G (2009) Optimal server resource reservation policies for priority classes of users under cyclic non-homogeneous markov modeling. Eur J Oper Res 198(2):545–556
18. Garg S, Moorsel A, Vaidyanathan K, Trivedi K (1998) A methodology for detection and estimation of software aging. In: Proceedings of 9th international symposium on software, reliability engineering, pp 282–292
19. Vaidyanathan K, Trivedi S (1999) A measurement-based model for estimation of resource exhaustion in operation systems. In: Proceedings of 10th international symposium on software, reliability engineering, pp 84–93
20. Vaidyanathan K, Trivedi S (2005) A comprehensive model for software rejuvenation. IEEE Trans Dependable Secur Comput 2(2):124–137
21. Cassidy K, Gross K, Malekpout A (2002) Advanced pattern recognition for detection of complex software aging in online transaxtion processing servers. In: Proceedings of dependable systems and networks, pp 478–482
22. Gross K, Bhardwaj V, Bickford R (2002) Proactive detective of software aging mechanisms in performance critical computers. In: Proceedings of 27th IEEE annual symposium on software enginerring, pp 17–23
23. Silva L, Alonso J, Torres J (2009) Using virtualization to improve software rejuvenation. IEEE Trans Comput 58(11):1525–1538
24. Matias R, Barbetta P, Trivedi K, Freitas P (2010) Accelerated degradation tests applied to software aging experiments. IEEE Trans Reliab 59(1):102–114

25. Avritzer A, Weyuker E (1997) Monitoring smoothly degrading systems for increased dependability. Empir Softw Eng J 2(1):59–77
26. Liu Y, Trivedi K, Ma Y, Han J, Levendel H (2002) Modeling and analysis of software rejuvenation in cable modem termination systems. In: Proceedings of 13th international symposium on software, reliability engineering, pp 159–170
27. Tai A, Chau S, Alkalaj L, Hecht H (1997) On-board preventive maintenance: analysis of effectiveness and optimal duty period. In: Proceedings of 3rd international workshop on object oriented real-time dependable systems, pp 40–47
28. Kourai K, Chiba S (2011) Fast software rejuvenation of virtual machine monitors. IEEE Trans Dependable Secur Comput 8(6):839–851
29. Wang D, Xie W, Trivedi K (2007) Performance analysis of clustered systems with rejuvenation under varying workload. J Perform Eval 64:247–265
30. Xie W, Shi Y, Xu G, Mao Y (2002) Smart Platform—a software infrastructure for smart space. In: Proceedings of 4th IEEE conference on multimodal, interfaces, pp 429–435